Doc. no.: X3J16/95-0201R4
WG21/N0801R4
Date: 30 January 1996
Project: Programming Language C++
Reply to: Beman Dawes
beman@dawes.win.net

Clause 17 (Library Introduction) Issues List - Version 4

History:

Initial: Distributed at the start of the Tokyo meeting.
Version 2: Distributed during the Tokyo meeting.
Version 3: Distributed in the post-Tokyo mailing. Reflects votes taken in Tokyo and issues added by the LWG sub-group.
Version 4: Distributed in the pre-Santa Cruz mailing.

<------------------------------------------------------------------------>

Work Group: Library Clause 17
Issue Number: 17-001
Title: Header Inclusion Policy
Section: 17.3.4.1 [lib.res.on.headers]
Status: Open
Description:

The header inclusion policy of allowing any C++ header to include any other C++ header results in portability problems with otherwise portable C++ programs.

The header inclusion policy of specifying exactly which C++ headers include which other C++ headers is difficult to specify correctly (due to the C++ library's complex dependency graph) and difficult to implement.

The WP currently (January 1996 troff) states:

> Certain types and macros are defined in more than one header. For such an entity, a second or subsequent header that also defines it may be included after the header that provides its initial definition (_basic.def.odr_).
>
> Header inclusion is limited as follows:
>
> --The C headers ( .h form, described in Annex D, _depr.c.headers_) shall include only their corresponding C++ header, as described above (_lib.headers_).
>
> --The C++ headers listed in Table 21, C++ Library Headers, shall include the header(s) listed in their respective Synopsis subclause ( [list of clauses] ).23)
>
> ___________________________
>
> 23) C++ headers must include a C++ header that contains any needed definition (_basic.def.odr_).

An editorial box contains:

> The header dependencies documented in this draft probably still contain errors. Worse, implementers may be overly constrained if they must restrict header inclusion only to the overt dependencies documented here. The Committee is actively exploring rules for header inclusion that are kinder to both users and implementers.

Resolution:

Requestor: Beman Dawes
Owner: Andrew Koenig

Emails:
Papers:

<------------------------------------------------------------------->

Work Group: Library Clause 17
Issue Number: 17-002
Title: Extending namespace std
Section: 17.3.3.1 [lib.reserved.names]
Status: Closed in Tokyo by accepting the proposed resolution.
Description:

The WP currently says (in 17.3.3.1 [lib.reserved.names]):

A C++ program shall not extend the namespace std.

Public comment:

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

Pre-meeting meeting comment:

>Needs editorial work in 17.3.3.1 to clarify under what conditions
>(if any) specializations of things already in a namespace are
>permitted. Also, this section does not address templates --
>what is allowed, etc. Need to address these cases:
>adding overloads; adding specializations.

The Clause 17 sub-LWG in Monterey felt that adding overloads was already covered by 17.3.4.4, but that specializations needed to be looked into and that the issue could turn out to be more that editorial.

Resolution:

Add the underlined wording to 17.3.3.1 [lib.reserved.names]):

A C++ program shall not extend the namespace std unless otherwise specified.

Add wording to 18.2.1 Numeric_limits [lib.limits]:

A program may extend the namespace std by supplying template specializations for numeric_limits.

Requestor: Public Comment 95-0136/N0736 T21-103 page 102
Owner:
Emails:
Papers:

<------------------------------------------------------------------->

Work Group: Library Clause 17
Issue Number: 17-003
Title: Violation of Requires preconditions
Section: 17.3.3 [lib.constraints]
Status: Closed in Tokyo by accepting the proposed resolution.
Description:

The effect of a program violating a library "Requires" paragraph are currently unspecified.

Resolution:

Add the following wording (presumably in a new section numbered 17.3.3.8) as part of 17.3.3 [lib.constraints]:

Violation of preconditions specified in a function’s "Requires" paragraph [lib.structure.specifications] results in undefined behavior unless the function’s "Throws" paragraph specifies throwing a exception when the precondition is violated.

Requestor: Beman Dawes
Owner:
Emails: Lib-3841, 3851.
Papers:

<---------------------------------------------------------------------->

Work Group: Library Clause 17
Issue Number: 17-004
Title: Should namespace std be subdivided?
Section: 17.3.1.1 Library contents [ lib.contents]
Status: Open
Description:

The original proposal (93-0136/N0343) adding namespaces to the standard library specified that each header would be in a separate namespace with the same name as the header and nested within namespace std.

These library sub-namespaces were later removed. It was felt that they added complexity without real benefits, and that much the same effect could be obtained by users via specific namespace-declarations.

Since that time the library has grown more complex. Several committee members feel that reintroducing sub-namespaces would result in a cleaner design for the library.

One case has been discovered where namespace-declarations cannot achieve the same effect as sub-namespaces:

```
using std::operator+;  // currently brings in all operator+'s

//  allowed if each header is in its own namespace within std:
using std::string::operator+;   // basic_string +
using std::numeric::operator+;  // valarray +
```

Resolution:

Modify the WP (17.3.1.1 Library contents [ lib.contents]) by adding a sentence after the sentence which reads:

> All library entities shall be defined within the namespace std.

The added sentence shall read:

> Within namespace std, all library entities shall be defined within a namespace with the same name as the header the entity is defined in.

Requestor: Randy Smith
Owner:
Emails: lib-4346, 4348, 4355, 4357, 4359, 4361, 4363, 4365, 4367
Papers:

<---------------------------------------------------------------------->

Work Group: Library Clause 17
Issue Number: 17-005
Title: What does “extending namespace std” mean?
Section: 17.3.3.1 [ lib.reserved.names]
Status: Open
Description:

The WP (as modified in Tokyo) says (in 17.3.3.1 [ lib.reserved.names]):

> A C++ program shall not extend the namespace std unless otherwise specified.

What does “extend” mean in 17.3.3.1?

Nathan Myers discusses the issues in message c++std-lib-4366:

In Tokyo we discussed the issue of what entities could be added by users to the "std" namespace. This is a little bit misleading, because users are not allowed to add names, or overload existing names, and we should be clear about that.

The issue is really about templates can be specialized in the std namespace. This is necessary in some cases because the user of the specialization cannot be expected to know that the type is specialized. In other cases it would be a substantial convenience.

Here are a few I (or others) have identified:

In Clause 25, all the algorithms must occasionally be specialized for certain data structures (or rather, their iterators). In Clause 18, the numeric_limits template must routinely be specialized for user types. Each of the containers in clauses 21 and 23 must be specialized for certain user allocators. Some of the iterator adaptors (e.g. insert) in Clause 24 must be (partial-) specialized for certain data structures. The traits template in Clause 27 must be specialized if users are to benefit by the default traits parameter on stream templates.

This list has got quite long, and my confidence in our ability to generate an exhaustive and correct list flags. I would like to propose instead that specializations of any standard template on any *user-defined type* are permitted. Then it becomes the responsibility of the definer of that type to ensure that any specializations implement the semantics described in the standard.

Resolution:

Modify 17.3.3.1 [lib.reserved.names] to read:

> A C++ program shall not add declarations or definitions to namespace std unless otherwise specified. A program may add template specializations for any standard library template to namespace std. Such a specialization (complete or partial) of a standard library template results in undefined behavior unless the declaration depends on a user-defined name of external linkage and unless the specialization meets the standard library requirements for the original template.

Add a footnote after 17.3.3.1 [lib.reserved.names] to read:

> Any library code that instantiates other library templates must be prepared to work adequately with any user-supplied specialization that meets the minimum requirements of the Standard.

Remove the wording 18.2.1 Numeric_limits [lib.limits] specifying that a program may extend namespace std with a numeric_limits specialization.

Requestor: John Spicer, Nathan Myers
Owner:
Emails:
lib-4366, 4368, 4370, 4371, 4373, 4375, 4376, 4377, 4380, 4381, 4385, 4394, 4398, 4400
Papers:

<--------------------------------------------------------------------->

Work Group: Library Clause 17
Issue Number: 17-006
Title: Action when program extends namespace std
Section: 17.3.3.1 [lib.reserved.names]
Status: Open
Description:

The WP (as modified in Tokyo) says (in 17.3.3.1 [lib.reserved.names]):

> A C++ program shall not extend the namespace std unless otherwise specified.

What happens when a program extends namespace std?

Resolution:

Option 1

Take no action.

Violation of “shall not” results in the program being ill-formed and thus requires the processor “issue at least one diagnostic error message.” See 1.7 [intro.compliance].

Option 2

Change 17.3.3.1 [lib.reserved.names] to read “It is undefined behavior for a C++ program to extend . . .”

It may be difficult for some implementations to diagnose illegal extension of namespace std. Making such extension undefined behavior gives implementations freedom to diagnose but does not require a diagnostic.

Requestor: John Spicer
Owner:
Emails:
Papers:

<--------------------------------------------------------------------->

Work Group: Library Clause 17
Issue Number: 17-007
Title: Which C++ headers have .h forms?
Section: 17.3.1.2 [lib.headers]
Status: Open
Description:

How should .h forms be specified for C++ headers?

Many existing C++ programs will be broken by the standard unless .h forms of headers are included for traditional C++ headers such as iostream.h.

The list of "C++ headers" that came with cfront 3.0 (and cfront 2.0 as it happens) was:

    common.h complex.h fstream.h iomanip.h iostream.h
    new.h stdiostream.h stream.h strstream.h task.h vector.h

Of these, the following still exist (as non .h headers) in the current (Sep 95) WP:

    fstream, iomanip, iostream, new, vector

The contents of <vector> is now completely different from that of cfront's <vector.h> and should be discounted, but it is reasonable to assume that existing code that uses the other .h headers from this list would want them to continue to exist and do approximately the same things.

Resolution:

Change list of C++ .h headers in the WP to include:

fstream.h
iomanip.h
iostreamh
new.h

(As of the Sept 95 WP, the relevant paragraph is in D.4 [depr.c.headers], but there is a request pending to move it to 17.3.1.2 [lib.headers].)

Requestor: Steve Rumsby
Owner: Steve Rumsby

Emails:
Papers:

<----------------------------------------------------------------------->

Work Group: Library Clause 17
Issue Number: 17-008
Title: Relational operator templates
Section: 20.2.1 [lib.operators]
Status: Open

Note: Treated as a Clause 17 issue due to wide impact.

Description:

The STL provides a set of templates defining the equality and inequality operators in terms of other operators in the set. The intention is that people defining their own classes to be used with the STL have to provide only some of the necessary operators, and the others will be "magically" provided. Currently, it is practically impossible to avoid having these operator templates apply to user written classes, even when they are not appropriate.

There are two obvious solutions to the problem. One is to remove the template operators altogether. The other is to put them in a namespace of their own. I believe that for this to work, no standard namespace should "use" this new namespace, or the operators will once again leak out, but I need to think further about this. The "Koenig lookup rule" and the transitive lookup rules we've added recently seem to make it harder to keep these operators under control, but I'm not sure.

Resolution:

Requestor: United Kingdom
Owner: Steve Rumsby
Emails:
Papers: 95-0170 = N0770 UK Comments Issue #135

<----------------------------------------------------------------------->

Work Group: Library Clause 17
Issue Number: 17-009
Title: Separate Library from Core Language in Document
Section:

Status: Open
Description:

The structure of the proposed reference manual shall be revised. We think that:

- The core language shall be separated from the libraries. The core language is what is handled by the compiler.
- Language support libraries should stay in the core language.
- Other libraries shall be in required (non-optional) normative annexes.

Resolution:

Take no action.

The current separation of the library into several clauses provides coherent organization. Moving some material to normative annexes would confuse the presentation. The document is intended to be an ANSI/ISO Standard conforming to IEC/ISO Directives - Part 3, Drafting and presentation. It is not intended to be a reference manual.

Requestor: France
Owner:
Emails:
Papers: 95/0199 = N0799 Issue R3

<---------------------------------------------------------------------->

Work Group: Library Clause 17
Issue Number: 17-010
Title: Too Many Classes and Features in Standard Library
Section:
Status: Open
Description:

The standard library contains far too many classes and features. It is clear that C++ (like C) needs a standard library. The reason for this is that the language itself has no method for communication with the system (I/O, date/time and other system dependencies). A standard library for these system-dependent features is necessary to allow portable code to be written. The C standard was focused upon these issues, however the proposed C++ standard library introduces many classes and features (in particular standard data structures and algorithms) that do not add to the portability of the language.

This is not to say that it would not be nice to have standard definitions for these features, but it should not be part of the C++ language standard. Adding these features to the CV++ standard complicates both the implementation (e.g. for embedded systems) and the learning of the language. Furthermore, it is not clear at the moment whether the features included in the library are really what is needed. In fact, if one considers the currently available commercial libraries as an indication. There is much more need for a standard library for handling GUI’s. This is understandable, since GUI interfaces are generally system dependent and a standard library for that would greatly improve portability. This cannot be said about e.g. a list data structure. A telling sign about the need for features of the current library is that currently it is not supported by any of the major C++ development environments.

Resolution:

Take no action.

The choice of classes and functions for the standard library involves complex tradeoffs between a desire to keep the library small and tractable, the need to provide a base set of functionality facilitating communication between third-party libraries, and many other factors.

The current library provides a set of classes and functions which has evolved over five years of detailed discussions, and the committee is satisfied that an acceptable balance has been struck between the competing needs.

Requestor: Netherlands
Owner:
Emails:
Papers: 95/0199 = N0799 Issue R19

<---------------------------------------------------------------------->

Work Group: Library Clause 17
Issue Number: 17-011
Title: Library Defined in Terms of Templates
Section:
Status: Open
Description:

NNI objects to the fact that the complete library has been defined in terms of templates. This makes it more difficult to use the library without a complete and in-depth knowledge of the language. A grow-path from C to C++ is also not facilitated by the use of templates in the library. Furthermore, it makes it impossible to implement the library on older systems (where templates might not be supported yet) and on smaller (e.g. embedded) systems (where templates are not implemented to save space).

Resolution:

Take no action.

Templates are heavily used in the standard library because they solve problems of internationalization and generic programming which were otherwise intractable. The pros and cons of template-based approaches were debated at great length and for several years during the development of the library, and the committee is satisfied that the issues have been fully aired and carefully considered.

Requestor: Netherlands
Owner:
Emails:
Papers: 95/0199 = N0799 Issue R20

<---------------------------------------------------------------------->

Work Group: Library Clause 17
Issue Number: 17-012
Title: Decouple Libraries
Section: 17
Status: Open
Description:

The general structure of the libraries shall be revised. In particular, it is necessary to decouple libraries. In the current proposal, there are unacceptable cross and forward references between libraries (and sections). The order of introduction of libraries is not adequate.

Resolution:

Make no specific changes in the WP as a result of this issue.

Work on certain other open issues, such as 17-001 (Header Inclusion Policy), will have the effect of both better specifying and reducing coupling between libraries. Certain other recent committee actions, such as the “decouple class exception” resolution passed in Austin (N0665R1 = 95-065R1), also reduced coupling between libraries.

Requestor: France
Owner:
Emails:
Papers: 95/0199 = N0799 Issue R4

<------------------------------------------------------------------->

Work Group: Library Clause 17
Issue Number: 17-013
Title: How will users access non-ISO C symbols using C++ headers?
Section: 17.3.1.2, Etc.
Status: Open
Description:

It seems that the intent of the draft 26-September-1995 Standard is to require C++ Standard library vendors to provide two sets of the 18 headers containing C Standard library facilities (as defined by ISO C and Amendment 1). One set named <cname> is provided to give C++ Standard library users access to ISO C features in the namespace std. This includes exactly 54 standard macros, 45 standard values, 19 standard types, 2 structures, and 208 standard functions (17.3.1.1). The C++ <cname> headers do contain some differences from ISO C but these are detailed in the draft. In line with this footnote 139 to section 17.3.4.2 (Restrictions on macro definitions) states that a global function can not be declared by the implementation as taking additional default arguments. The other set, named <name.h> is provided for compatibility only and is deprecated C++ functionality (D.4). The draft states that C++ provided <name.h> headers should only include their corresponding C++ <cname> header followed by an explicit using-declaration for each name placed in the Standard library namespace by the header (17.3.4.1, D.4).

This scheme raises the issue of how C++ users access non-ISO C symbols through the C++ header files. Non-ISO C symbols include symbols placed in these headers by the various ISO POSIX standards, X/Open CAE Specification, Issue 4, Version 2, as well as vendor specific extensions. In addition it adds confusion by supporting two sets of ISO C headers (one set provided by C++ as per D.4 and one set provided by the C language).

There are many examples of commonly used ISO C extensions in the 18 C++ Standard library headers provided for C library compatibility which would not be available to C++ users (even in the global namespace) if <cname> and <name.h> C++ headers do not allow extensions. Use of the Standard C library localtime(), ctime(), strftime() and mktime() functions whose prototypes will exist in the C++ supplied <ctime> and <time.h> headers offer one example. Often the tzset() function (defined by the ISO POSIX standards as well as the X/Open CAE Specification, Issue 4, Version 2) is used in conjunction with the time routines to set and access timezone conversion information. Since tzset() is not defined by ISO C, it would not be provided by the C++ Standard library. A C++ user including <ctime> and/or a C++ version of <time.h> would not have access to the prototype for tzset().

Similarly consider the confusion which arises in the following example of a C++ program which includes a C library header say newlibc.h which includes time.h in order to make visible the POSIX symbol CLK_TCK. If the C++ Standard library headers of the form <name.h> do not provide access to ISO C extensions then what will be visible to the C++ program including newlibc.h? Will newlibc.h include the C++ Standard library version of time.h which only includes <ctime> which does not define CLK_TCK? Or will it include the existing C library header which contains the symbol?

Resolution:

A set of changes to Sections 17.3 (Library-wide requirements) and subclause D.4 (Standard C library headers) of the draft 26-September-1995 C++ Standard are proposed to clarify the use of non-ISO C extensions in the C++ headers for C library facilities. The proposed changes follow:

1. Section 17.3.1.2 (Headers) paragraph 4 states:

    Except as noted in Clauses 18 through 27, the contents of each header cname shall be the same as that of the corresponding header name.h, as specified in ISO C (Clause 7) or Amendment 1, (Clause 7), as appropriate. In this C++ Standard library, however, the declarations and definitions are within namespace scope (3.3.5) of the namespace std.

    Proposed Wording:

    Except as noted in Clauses 18 through 27, the contents of each header cname shall be the same as that of the corresponding header name.h, as specified in ISO C (Clause 7) or Amendment 1, (Clause 7), as appropriate. In this C++ Standard library, however, the declarations and definitions from ISO C (Clause 7) or Amendment 1 (Clause 7) are within namespace scope (3.3.5) of the namespace std.

    Rationale:

    This change acknowledges the existence of non-ISO C symbols in the 18 C++ Standard library headers for C library facilities. It specifies that ISO C declarations and definitions are provided within namespace scope of the namespace std. Non-ISO C symbols may be provided but not within the namespace scope of the namespace std.

2. Section 17.3.4.1 (Headers) paragraph 2 line 1 states:

    -- The C headers (.h form, described in Annex D, D.4) shall include only their corresponding C++ header, as described above (17.3.1.2)

    Proposed Wording:

    Strike this section as it is already covered in section D.4 which is discussed below.

    Rationale:

    Keep the discussion of the C++ Standard library <name.h> headers in one place.

3. Section 17.3.4.2 (Restrictions on macro definitions) Footnote 139 states:

    A global function cannot be declared by the implementation as taking additional default arguments. Also, the use of masking macros for function signatures declared in C headers is disallowed, notwithstanding the latitude granted in subclause 7.1.7 of the C Standard. The use of a masking macro can often be replaced by defining the function signature as inline.

    Proposed Wording:

    A global function in the std namespace cannot be declared by the implementation as taking additional default arguments. Also, the use of masking macros for function signatures declared in C headers is disallowed in the std namespace, notwithstanding the latitude granted in subclause 7.1.7 of the C Standard. The use of a masking macro can often be replaced by defining the function signature as inline.

    Rationale:

This change acknowledges the existence of non-ISO C symbols in the 18 C++ Standard library headers for C library facilities. It specifies that ISO C declarations and definitions are provided within namespace scope of the namespace std. Non-ISO C symbols may be provided but not within the namespace scope of the namespace std.

4. Section D.4 (Standard C library headers) paragraph 2 states:

Each C header, whose name has the form name.h includes its corresponding C++ header cname, followed by an explicit using-declaration (7.3.3) for each name placed in the standard library namespace by the header (17.3.1.2).

Proposed Wording:

Each C header, whose name has the form name.h behaves as if each name placed in the Standard library namespace by the corresponding cname header is also placed within the namespace scope of the namespace std and is followed by an explicit using-declaration (7.3.3).

Rationale:

This wording specifies the intent of the <name.h> headers without specifying the implementation. It allows C++ Standard library vendors to supply the <name.h> headers listed in D.4 in the way that is most appropriate to their users.

Requestor: Sandra Whitman, DEC
Owner:
Emails: Lib-4348
Papers:

<------------------------------------------------------------------------>

Work Group: Library Clause 17
Issue Number: 17-014
Title: Requirements on compare functions
Section: Various
Status: Open
Description:

In the reflector message listed below, “some problems with the requirements on comparison objects” are discussed and solutions provided, including suggested working paper wording.

The suggested changes apply to several library clauses including 20, 23, and 25. The purpose of this issue 17-014 is simply to insure that all of the issued raised get dealt with as appropriate.

Resolution:

Requestor: Dave Musser
Owner:
Emails:
lib-4386 Requirements on compare functions
lib-4387 Complete list of changes related to compare functions
lib-4402 Re: Complete list of changes related to compare functions
lib-4403 Re: Requirements on compare functions
Papers:

<------------------------------------------------------------------------>

Work Group: Library Clause 17
Issue Number: 17-015
Title: Restrictions on macro definitions clarification

```
Section:       17.3.4.2
Status:        Open
Description:

In section 17.3.4.2  (Restrictions on macro definitions) footnote 139
    states that the use of masking macros for function signatures
    declared in C headers is disallowed.  I would like some clarification
    about what a masking macro is.  An example of such a macro in the
    footnote would help explain its use.

Resolution:

    Editorial. Forward to Editor for clarification.

Requestor:     Judy Ward
Owner:
Emails:
Papers:
```