Doc. No.: X3J16/96-0117
WG21/N0935
Date: May 22, 1996
Project: Programming Language C++
Reply To: Sandra Whitman
Digital Equipment Corporation
whitman@tle.enet.dec.com

Clause 18 (Language Support Library) Issues List - Version 4

Revision History

Version 1 - February 1, 1995: Distributed in pre-Austin mailing.
Version 2 - May 30, 1995: Distributed in pre-Monterey mailing.
Version 3 - September 26, 1995: Distributed in pre-Tokyo mailing. Closed issues are compressed to save paper.
Version 4 - May 22, 1996: Distributed in pre-Stockholm mailing.

Introduction

This document is a summary of the issues identified in Clause 18. For each issue the status, a short description, and pointers to relevant reflector messages and papers are given.

Active Issues

Work Group: Library Clause 18
Issue Number: 18-015
Title: Should terminate() and unexpected() be in <exception> ?
Sections: 18.6 Exception handling [lib.support.exception]
18.6.2.4 unexpected [lib.unexpected]
18.6.3.3 terminate [lib.terminate]
Status: active
Description: Nathan Myers in a private mail:

> [The discussion is why terminate() and unexpected() are declared in <exception>. I had speculated:]
>
> > > 1. They are present so that users can call them to simulate
> > > the event normally generated only by the runtime environment.
> > > 2. They are present so that users can restore the original behavior,
> > > even if they didn't originally call set_*_handler.
> > > 3. They are present so their address can be compared against
> > > the result of calling set_*_handler.
> > >
> [spicer replied:]
> > Of these, I believe that only #1 is possible. The default terminate
> > handler is not terminate(), but rather an implementation defined
> > function that calls abort(). If you were to do
> >
> > set_terminate(&terminate);

>
> you would probably end up with an infinite loop (until you ran out
> of stack space). For the same reason, a call to set_terminate would
> never return the address of terminate() as the previous handler value.
> The same applies to unexpected.
>
> It seems odd to permit #1, particularly for unexpected. I would actually
> prefer that it be undefined if a user calls either of these
> functions.

This is worth bringing up in the Lib WG. I suspect
that we didn't really look closely enough at this and just assumed
as I did that unexpected() was itself the default handler.

If these functions aren't mentioned in a header file, and can't
be called by users, why mention them at all? On the other hand,
wouldn't it be simpler if they were just the default handler?

Proposed Resolution:

Remove terminate() and unexpected() from <exception>

Change clause 18.6 Exception handling [lib.support.exception]
as follows:

1. remove void unexpected(); void terminate(); from
   <exception> synopsis.

2. check usage in 18.6.2.2, 18.6.2.4, 18.6.3.1, 18.6.3.3,
   8.6.4

Requestor: Nathan Myers, ncm@cantrip.org
Owner: Sandra Whitman
Emails: c++std-lib-4725, 4728
Papers: None.

Work Group: Library Clause 18
Issue Number: 18-016
Title: numeric_limits and LIA-1/WG14/C Compliance
Sections: 18.2.1 Numeric limits [lib.limits]
Status: active
Description: Nathan Myers in a private email:

Someone needs to do some real analysis here. There are quite a few open
issues:

1. Are we REQUIRED to be LIA-1 compliant?
2. What are they doing in WG14 in this area?
3. How do we keep compatibility with C? Is it possible?
4. Is it enough to add a few new members to numeric_limits, or do we need
   to add a whole bunch of extra stuff (LIA-1, Annex E.4 suggests a
   <lia.h> header for C implementations wishing to comply to LIA-1).

Proposed Resolution:

Complete analysis required to provide a solution to the problem of
LIA-1 conformance.

Requestor: Nathan Myers, ncm@cantrip.org
Mike Lijewski, lijewski@roguewave.com
Owner: Sandra Whitman
Emails: c++std-all-1262 mentions LIA-1.
c++std-lib-3975.
Papers: Suggested reading is ISO/IEC 10967-1:1994.
(IEC 559 is the same as IEEE 754, and it is a subset of
"ISO/IEC 10967-1, Language independent arithmetic -
Part 1: Integer and floating point arithmetic"
(also known as LIA-1).

Work Group: Library Clause 18
Issue Number: 18-017
Title: Run-time Dependent traps in numeric_limits
Sections: 18.2.1 Numeric Limits [lib.support.limits]
Status: active
Description: Mike Lijewski in c++std-lib-3975:

> I can imagine an implementation where
> the value of numeric_limits<double>::traps depends on the setting
> of some user-settable math library flags; i.e. the value of
> numeric_limits<double>::traps could be true in one part of a
> program and false in another, depending on what, if any,
> OS-specific math library calls the user's made. In any case, I
> don't see a good reason why this should be precluded.

The problem here is that changing this member to be an inline static
(member) function would impose a performance overhead.

Proposed Resolution:

Change numeric_limits<T>::traps to an inline static member function.

Requestor: Mike Lijewski, lijewski@roguewave.com
Owner: Sandra Whitman
Emails: c++std-lib-3975.
Papers: Suggested reading is ISO/IEC 10967-1:1994.

Work Group: Library Clause 18
Issue Number: 18-018
Title: Run-time Dependent Rounding in numeric_limits
Sections: 18.2.1 Numeric limits [lib.limits]
Status: active
Description:

There are systems where the rounding style for floating point numbers
isn't constant. This member:

numeric_limits<float|double|long double>::round_style

can be changed by calling the IEEE function fpsetround at run time. Additionally if the initial rounding style is set by the run-time environment, the initializer for round_style isn't a constant expression as it can only be determined by calling fpgetround and related functions. (SDW 5/96, I believe these are equivalent to the fesetround/fegetround functions described by WG14/N319, X3J11/94-003 Floating-Point C Extensions)

Proposed Resolution:

1. Add a new enum value to "18.2.1.3 [lib.round.style]":

```
namespace std {
  enum float_round_style {
    round_indeterminate       = -1,
    round_toward_zero         =  0,
    round_to_nearest          =  1,
    round_toward_infinity     =  2,
    round_toward_neg_infinity =  3,
    round_runtime_dependent   =  4      // New enum value
  };
}
```

2. Add a new inline static (member) function to "18.2.1.1 [lib.numeric.limits]":

```
namespace std {
  template<class T> class numeric_limits {
  public:
    // Current list
    static float_round_style current_round_style() throw();  // New
  };
}
```

   This function shall return the current round style, and may therefore not return float_round_style::round_runtime_dependent.

3. It should also be added in the text that these members are meaningful for floating points only.

   The text for 2 and 3 above in 18.2.1.2 could be (SDW 5/96):

   static float_round_style current_round_style() throw();

   Dynamic rounding mode, if available. May not return float_round_style::round_runtime_dependent. (SDW 5/96, can an error be returned by this routine?)

   Meaningful for floating point types which adhere to IEC 559.

Requestor: Dominik Strasser, Dominik.Strasser@mch.sni.de
Owner: Sandra Whitman
Emails: c++std-lib-4073, 4091
Papers: Suggested reading is ISO/IEC 10967-1:1994.
Discussion:

It was difficult to select a good name for the new enum value. Dominik and I had at least this list to choose from:

```
       round_runtime_dependent    // Selected
       round_varying
       round_variable
       round_fluctuate
       round_runtime_determinable
       round_volatile
       round_non_constant
```

Someone fluent in English might have objections to the suggested name.

Work Group: Library Clause 18
Issue Number: 18-019
Title: Extra Denorm Members in numeric_limits in Support of IEC 559
Sections: 18.2.1 Numeric limits [lib.limits]
Status: active
Description: Nathan Myers in a private email:

In support of iec559 there should be two or three other parameters describing denormalized number behavior.

Proposed Resolution:

Add additional denorm members. (Details from Nathan needed)

Requestor: Nathan Myers, ncm@cantrip.org
Owner: Sandra Whitman
Emails: c++std-all-1262 mentions LIA-1.
Papers: Suggested reading is ISO/IEC 10967-1:1994.

Work Group: Library Clause 18
Issue Number: 18-020
Title: numeric_limits static const int/bool Members Must be Constant Expressions.
Sections: 18.2.1 Numeric limits [lib.limits]
Status: active
Description: Nathan Myers in c++std-lib-4594

The default definition of the template numeric_limits<> is still not right. It's important for the int and bool static const members to be compile-time constants, both in the default definition and in any vendor or user specializations. That is, members should look like:

```
   static const int digits = 0;
```

not

```
   static const int digits;
```

This makes a difference because user code can say for example:

```
  char digits[numeric_limits<T>::digits + 1];
```

or

```
  case numeric_limits<T>::digits:
```

which would not compile if it were an out-of-line constant. The original proposal specified things this way (and no proposal changed it) but editorial tinkering has stripped off the definitions.

Proposed Resolution:

1. In the class template declaration in [lib.numeric.limits], for all static const integral or enumerated members:
   add " = 0" int members
   add " = false" to bool members
   add " = round_toward_zero" to the member round_style.

   So in 18.2.1.1 numeric_limits would look like this:

```
template<class T> class numeric_limits {
public:
  static const bool is_specialized = false;
  static T min() throw();
  static T max() throw();
  static const int digits = 0;
  static const int digits10 = 0;
  static const bool is_signed = false;
  static const bool is_integer = false;
  static const bool is_exact = false;
  static const int radix = 0;
  static T epsilon() throw();
  static T round_error() throw();
  static const int min_exponent = 0;
  static const int min_exponent10 = 0;
  static const int max_exponent = 0;
  static const int max_exponent10 = 0;
  static const bool has_infinity = false;
  static const bool has_quiet_NaN = false;
  static const bool has_signaling_NaN = false;
  static const bool has_denorm = false;
  static const bool has_denorm_loss = false;
  static T infinity() throw();
  static T quiet_NaN() throw();
  static T signaling_NaN() throw();
  static T denorm_min() throw();
  static const bool is_iec559 = false;
  static const bool is_bounded = false;
  static const bool is_module = false;
  static const bool traps = false;
  static const bool tinyness_before = false;
  static const float_round_style round_style = round_toward_zero;
};
```

2. Add a paragraph to 18.2.1.1:

   For all members declared "static const" in the template above, specializations must define these values in such a way that they are usable as constant expressions.

Requestor: Nathan Myers, ncm@cantrip.org
Owner: Sandra Whitman
Emails: c++std-lib-4594,4596,4597,4639
Papers: None

Work Group: Library Clause 18
Issue Number: 18-021
Title: Correction to nothrow in <new>
Sections: 18.4 Dynamic memory management [lib.support.dynamic]
Status: active
Description: John Spicer in a private email:

> > >I think there is a minor problem with the proposed change.
> > >
> > >I believe that
> > >
> > > const nothrow_t nothrow;
> > >
> > >should be changed to
> > >
> > > const nothrow_t nothrow = {};
> > >
> > >because const objects must be initialized.
> >
> > Thanks, John.
> >
> > Several people want it changed to:
> >
> >   enum nothrow_t { nothrow };
> >
>
> I take it that the objection to the original proposal was that
> people didn't like having a "nothrow" object allocated in each
> translation unit where it was used? If so, why not just require that
> the library define the object and just have a declaration in the
> header file?
>
> I can think of two potential problems with the enum approach:
>
> 1. There is an implicit conversion from enum to int, so nothrow will
>   match an integral argument (although the one taking an enum is
>   preferred).
>
> 2. The declaration given above gives nothrow the value zero,
>   which will also match any pointer type argument as it is a
>   null pointer constant. As with point #1, the enum version is
>   still preferred.
>
> Why is this a problem, if the enum version is preferred?
> Because it makes writing class specific operator new functions
> more error-prone. The following example calls the class specific
> placement new because the user forgot to supply a nothrow version.
> In error message would be a much better result.
>
> John.
>
> typedef unsigned int size_t;
> enum nothrow_t { nothrow };
>
> struct A {
>   void* operator new(size_t, void*); // placement new
> };

```
    >
    > int main()
    > {
    >         A* ap = new (nothrow) A;  // calls placement new
    > }
    >
```

Proposed Resolution:

Change:

```
    struct nothrow_t{};
    const nothrow_t nothrow;
```

To (choose one):

```
    1) struct nothrow_t{};
       const nothrow_t nothrow = {};

    2) enum nothrow_t { nothrow };

    3) struct nothrow_t {};
       extern nothrow_t nothrow;  // defined in library
```

Requestor: John Spicer, Jerry Schwarz
Owner: Sandra Whitman
Emails: c++std-lib-4725, 4728
Papers: None

Work Group: Library Clause 18
Issue Number: 18-022
Title: Make nothrow a Type Instead of a Value.
Sections: 18.4 Dynamic memory management [lib.support.dynamic]
Status: active
Description: Clause 18-editorial box 1

Currently section 18.4 contains an editorial box which states:

The division of labor between the global namespace and namespace std should probably be reexamined, as should making nothrow a type instead of a value. ARK 9/95

The issue of making nothrow a type was addressed at the Santa Cruz meeting. It is additionally addressed by 18-021.

The issue of global namespace verses std namespace may need further clarification. (May have been addressed by 18-008)

Proposed Resolution:

Remove Box 41 (make sure that the namespace issue is closed).

Requestor: Sandra Whitman
Owner: Sandra Whitman
Emails: None
Papers: None

Work Group: Library Clause 18
Issue Number: 18-023
Title: Array Form of Operator delete[] Added to 18.4.1.2
Sections: 18.4.1.2 Array forms [lib.new.delete.array]
Status: active
Description: Clause 18-editorial box 2

Currently section 18.4.1.2 contains an editorial box which states:

The array form void operator delete[] (void* ptr, const std::nothrow&) throw(); was added during editing to correct an oversight in issue 18-014. BGD 1/96

Since 18-014 has been closed this box should be removed.

Proposed Resolution: Remove Box 42
Requestor: Sandra Whitman
Owner: Sandra Whitman
Emails: None
Papers: None

Work Group: Library Clause 18
Issue Number: 18-024
Title: Are Some numeric_limits static const Members Really Dynamic ?
Sections: 18.2.1 Numeric limits [lib. limits]
Status: active
Description: Daveed Vandevoorde in c++std-lib-4637

c++std-lib-4637 suggests that some of the static constant members in numeric_limits might be dynamic.

> Aren’t some of these constants are not so constant in practice?
> I believe the rounding style for example can be set at run-time
> on several platforms.

(SDW 5/96) 18-017 proposes replacing the static const bool traps member with a static traps routine. 18-018 proposes adding a routine to provide a runtime rounding mode. Other static const numeric_limits members may fall into this category.

Proposed Resolution:

Determine if any static const numeric_limits members really require runtime support.

Requestor: Daveed Vandevoorde
Owner: Sandra Whitman
Emails: None c++std-lib-4594,4596,4597,4639
c++std-lib-4637
Papers: None

Work Group: Library Clause 18
Issue Number: 18-025
Title: Make references to throw references to throw() in 18.2.1
Sections: 18.2.1 Numeric limits [lib. limits]
Status: active
Description: Editorial; throw should be throw() in 18.2.1
Proposed Resolution: Change throw to throw() in 18.2.1
Requestor: Sandra Whitman
Owner: Sandra Whitman
Emails: None
Papers: None

Closed Issues

Issue Number: 18-001
Title: Typedef typedef void fvoid_t(); not used anywhere
Last Doc.: N0784=95-0184

Issue Number: 18-002
Title: Redundant typedefs
Last Doc.: N0784=95-0184

Issue Number: 18-003
Title: Call to set_new_handler() with null pointer
Last Doc.: N0784=95-0184

Issue Number: 18-004
Title: Inherited members explicitly mentioned
Last Doc.: N0784=95-0184

Issue Number: 18-005
Title: Call to set_terminate() or set_unexpected() with null pointer
Last Doc.: N0784=95-0184

Issue Number: 18-006
Title: <stdarg.h> and references
Last Doc.: N0784=95-0184

Issue Number: 18-007
Title: denormal_loss member to the numeric_limits class
Last Doc.: N0784=95-0184

Issue Number: 18-008
Title: global operator new
Last Doc.: N0784=95-0184

Issue Number: 18-009
Title: whither exception?
Last Doc.: N0784=95-0184

Issue Number: 18-010
Title: Exception specifications for class numeric_limits
Last Doc.: N0784=95-0184

Issue Number: 18-011
Title: Exception specifications for set_new_handler()
Last Doc.: N0784=95-0184

Issue Number: 18-012
Title: Exception specifications for set_unexpected() and set_terminate()
Last Doc.: N0784=95-0184

Issue Number: 18-013
Title: deleting a pointer obtained by a nothrow version of
"operator new"
Last Doc.: N0784=95-0184

Issue Number: 18-014
Title: nothrow versions of "operator delete"
Last Doc.: N0784=95-0184