Doc No: X3J16/97-0106
WG21/N1144
Project: Programming Language C++
Date: November 14, 1997
Reply to: Stephen D. Clamage
stephen.clamage@eng.sun.com

CD2 - Response to US Public Comments

<<<<<< Public Comment #01/DeRocco" follows >>>>>>
Date: January 10, 1997

I know it's been discussed, but I'd like to add my encouragement for a variant of #include that automatically suppresses previously included files. I'd suggest calling it #header. It should be required to detect the duplicate if the names pass a straight string comparison, case insensitive if that's appropriate for the OS, and if they're found in the same directory in the search path. Anything fancier shouldn't be required.

--

Paul D. DeRocco
DeRocco Engineering
87 Duff St.
Watertown, MA 02172
617-923-8987
mailto:pderocco@ix.netcom.com

RESPONSE: Previously considered and rejected. Rejected on this occasion as request for an extension.

<<<<<< Public Comment #02/DeRocco" follows >>>>>>
Date: January 10, 1997

Currently, an empty exception-specification merely indicates that a function will not throw an exception. There are two possible, and competing, interpretations of this. Either it's a command that the function must be prevented from throwing an exception, thus requiring the compiler to add code to trap any that do occur; or it's a hint that the function cannot possibly throw an exception anyway, so the compiler can take out code that deals with exceptions.

These compete in that if you care about efficiency, you need to know which it means. If it means the latter, you will be eager to tack "throw()" onto all those functions that won't throw exceptions, because doing so will potentially make the program more efficient. If it means the latter, you will be reluctant to, because doing so will do just the opposite.

My compiler (Borland) interprets it in the former manner, which seems reasonable, but isn't particularly useful to me. I'd like to suggest an additional syntax, namely "throw(void)", that would be defined as a hint that callers could assume no exceptions. Compilers would be allowed to ignore this specification, but would be prohibited from enforcing it.

Here's a situation where it could aid optimization:

```
unsigned func(unsigned a, unsigned b) throw(void) {
      return a + b; // really can't throw any exceptions
}
class foo {
      // ...
      ~foo(); // has a destructor
};
void test() {
      foo f; // create a foo
      // ...
      x = func(y, z); // can't throw anything
      // ...
} // destroy the foo
```

In the absence of the empty exception-specification, an exception context would need to be set up in order to guarantee the destruction of f. So, I'd like to put "throw()" onto the function to eliminate this overhead. However, with my current compiler (and probably most compilers), this results in far more inefficiency inside func() than it saves inside test(), so it's not worth it. I therefore never use "throw()". If I had "throw(void)" as an alternative construct, I'd actually use it, and it would do some good.

--

Paul D. DeRocco
DeRocco Engineering
87 Duff St.
Watertown, MA 02172
617-923-8987
mailto:pderocco@ix.netcom.com

***RESPONSE*:**

part 1 answer to question:

    If a function with a throw() exception specification throws an exception, unexpected() (15.5.2 except.unexpected) is called.

part 2 rejected -- request for an extension.

<<<<<< Public Comment #03/DeRocco" follows >>>>>>

Date: January 10, 1997

One of the things that C and C++ lacks that is hard to work around is fractional numeric types. These are essential in non-floating-point DSP algorithms. In particular, it is important to be able to do fractional multiplication, where the high half of the result is used instead of the low half. The only way to do fractional arithmetic currently is to use in-line assembler, which is non-portable, or function calls to assembly language library routines, which are inefficient.

I'd like to suggest a "fractional" modifier that could be applied to any numeric type (except "bool") which would cause it to be interpreted as having a [0,1) or [-0.5,0.5) range. There would be no conversions between integers and fractions, but there would be the obvious ones between floats and fractions. The only arithmetic operations that would allow integers and fractions to be combined would be:

```
      i = i * f         f = f * i
      i = i / f         f = f / i
```

The only oddity is that i * f would have a different type from f * i. The usual rules for promoting shorter types to longer ones would apply, except that numbers would be lengthened by zero-filling on the right. Also, as usual, unsigned would override signed.

I would suggest that casting from a fraction to an integer or vice versa would simply reinterpret the bits, since there is no other useful conversion.

There would be no need to introduce a new kind of literal. Fractional literals could be written by casting float literals.

--

Paul D. DeRocco
DeRocco Engineering
87 Duff St.
Watertown, MA 02172
617-923-8987
mailto:pderocco@ix.netcom.com

RESPONSE: rejected -- request for an extension.

<<<<<< Public Comment #04/DeRocco" follows >>>>>>
January 10, 1997

I'd like to suggest a new operator, a colon, perhaps called the "else" operator. The expression x:y would be defined as x?x:y, except that x would never be reevaluated if it is true. This is actually very useful in cases where x either has side effects or is textually long, and it wouldn't break any existing code. Since ?: isn't overloadable, : wouldn't be either.

One could also imagine a "then" operator consisting of a question mark. In other words, x?y would be defined as x?y:0. This isn't nearly as useful, though, since :0 is pretty easy to write, and doesn't provide any additional optimization. Besides, if there's a binary : operator, having a binary ? operator would make the ternary ?: operator ambiguous. Perhaps it would have been nice if && and || worked this way from the beginning. That is, x&&y could have been x?y:0, and x||y could have been x?x:y without reevaluating x. But it's too late for that. However, a simple binary : operator would be pretty easy to add.

--

Paul D. DeRocco
DeRocco Engineering
87 Duff St.
Watertown, MA 02172
617-923-8987
mailto:pderocco@ix.netcom.com

RESPONSE: rejected -- request for an extension.

<<<<<< Public Comment #05/DeRocco" follows >>>>>>
Date: January 10, 1997

Here are a few of _trivial_ additions to the token syntax that wouldn't break any existing C++ programs, but would increase clarity:

1) Allow binary numbers to be written with a 0y prefix. A bit mask like 0y11011111 is much easier to comprehend than 0xDF. (One might prefer 0b to 0y, because y might be mistaken for x, but b might be mistaken for a hex digit. I like y because it is the last letter of binary, just as x is the last letter of hex.)

2) Allow embedded underscores within numbers, which would be ignored. 1_000_000_000 is a lot clearer than 1000000000, and 0y1111_1000_0111_1111 is a lot clearer than 0y1111100001111111.

3) In a string or character constant, define \e as an escape, since it's such a common control character.

These are so easy and harmless that I see no excuse for not putting them in.
--
Paul D. DeRocco
DeRocco Engineering
87 Duff St.
Watertown, MA 02172
617-923-8987
mailto:pderocco@ix.netcom.com

RESPONSE: rejected -- request for an extension.

<<<<<< Public Comment #06/Lilley" follows >>>>>>
January 16, 1997

Hello,

This is a summary of a recent newsgroup discussion from comp.std.c++, in which we could find no definitive language to answer the question "is a class template instantiated as a result of calling delete on a pointer to an incomplete template specialization?" There is a related question concerning operator&() when applied to a reference-to-incomplete-template-specialization.

Let's start with an example:

```
template <class T> class A {
      A* operator&() { return this; }
public:
      void operator delete(void*) { ... }
};
void f(A<int>* ap) {
      delete ap; // #1
      A<int>* ap2 = &(*ap); // #2
}
```

The two relevant questions are:

On #1, is A<int>::operator delete() instantiated, or the default operator delete used, leading to undefined behavior?

On #2, is A<int>::operator&() instantiated (leading to an access violation), or is the default operator&() used?

The following language suggests the "undefined behavior" interpretation for both #1 and #2:

With regards to #1:
section 5.3.5, para 5
"If the object being deleted has incomplete class type at the point of deletion and the complete class has a non-trivial destructor or a deallocation function, the behavior is undefined."

With regards to #2:
section 5.3.1, para 4:
"The address of an object of incomplete type can be taken, but if the complete type of that object is a class type that declares operator&() as a member function, then the behavior is undefined..."

However, there is some additional language that suggests that the template should be instantiated:
section 14.7.1, para 3:

"If a class template for which a definition is in scope is used in a way that involves overload resolution, conversion to a base class, or pointer-to-member conversion, the class template specialization is implicitly instantiated."

section 13.3, para 2:
"Overload resolution selects the function call in seven distinct contexts within the language... --invocation of the operator referenced in an expression..."

section 13.3.1.2, para 2:
"If either operand has a type that is a class or an enumeration, a user-defined operator function might be declared that implements this operator... In this case, overload resolution is used..."

It seems to me that deleting an object involves overload resolution, because the operator delete is overloaded at the global scope and can be overloaded at the class scope. However, this is not completely supported by the language of 13.3.1.2/2, because the operand to delete is not of class type but of type pointer-to-class. It seems that it should be clear from other contexts that operator delete does indeed involve overloading, but the explicit language does not confirm that conclusion.

The case for instantiating a template when operator&() is involved is more clear, because operator& is applied to an operand of type class.

In my opinion, given that comp.std.c++ could reach no definite conclusion given the existing draft language, some explicit clarification on these matters is desirable.

Thank you for your consideration.

John Lilley
jlilley@empathy.com
Nerds for Hire, Inc.
4270 Evans Dr.
Boulder, CO 80303

RESPONSE: Accepted. A class template is instantiated if a pointer to a specialization is the operand of delete.

<<<<<< Public Comment #07/Lilley" follows >>>>>>
January 18, 1997

Hello,
I have been recently attempting the implementation of templates in my parser, and have found that the dec96 draft has not completely cleared upsome issues concerning the definition of "dependency on a template parameter".

The last draft I obtained (may96) contained side-comments to the effect that additional clarification was needed for the exact definition of "depends on", and I notice that some additional work has been done. I am comfortable with the current definition of dependency (14.6.2.1, 14.6.2.2), but not with its use in the examples illustrating ill-formed templates resulting from such dependency (14.6.2/2, 14.6.2/3). To the contrary, I believe that conclusions drawn about the examples given are unsupportable.

Consider the example in 14.6.2/3. It is claimed that the call g(1) is dependent on the template type argument only when the type argument is "int". However, this is not supported by the definition of dependency. 14.6.2/1 claims that a postfix expression such as g(1) is dependent on a type parameter "if and only if any of the expressions in the epression-list is a type dependent expression", which I take to mean that the literal "1" must be type-dependent for g(1) to be type-dependent. The unintuitiveness of this idea aside, 14.6.2.2/5 explicitly says that a literal is never type-dependent, so that example is incorrect.

The problem is that the examples given demonstrate a dependency on a template *argument*, whereas the dependency rules define dependency in terms of template *parameters*. Dependency on a template argument implies that any declaration or expression involving use of a type that *happens to be the same* as a type used as a template argument is fair game for being rendered "dependent" (or not) at the time of template instantiation. This in turn implies that all declarations may be rebound during the instantiation, given the right context.

Dependency on a template-argument cannot be defined statically because one cannot know what is dependent on a template-argument until one sees the template argument used in the instantiation -- dependency, in that context, is not a static property of the template declaration, but rather a property of the environment in which the template is instantiated, combined with the template declaration.

In my opinion, the best solution is to completely remove dependency on a template *arguments* and limit dependency to template *parameters*. That would make it possible to determine statically which names and expresions are dependent. This is especially true considering the new language of 14.6.2/5 concerning dependent base classes.

If this were adopted, then the practical result would be that no declaration or expression could be considered dependent unless it depends on the formal template parameter, and this in turn is known when the template declaration is processed. The fact that an expression or declaration involves a type that *happens to be the same as* one of the template arguments encountered later would not be considered.

This may indeed be the intent of the dec96 draft -- perhaps the examples that I referred to were not intended to conflict with other language.

Note that my suggestion would still allow for overloading of functions declared after the template, as long as the call to the overloaded function involved a template parameter. For example:

```
void g(long);
template <class T> class A {
      T t;
      void f() {
            g(t); // dependent on T
      }
}
void g(int);
A<int> ai;
ai.f(); // calls g(int);
```

If this were, done, then the rules concerning when a name may or may not be re-bound would be much clarified.

respectfully submitted,
John Lilley
jlilley@empathy.com
Nerds for Hire, Inc.
4270 Evans Dr.
Boulder, CO 80303

RESPONSE: Accepted. The second phase of name look up in template definitions applies to names dependent on the template parameters.

<<<<<< Public Comment #08/Lilley" follows >>>>>>
January 20, 1997

Hello,

I have some more comments regarding dependency on a template parameter whenthe template parameter is involved in the base class. 14.6.2/5 clarifies dependency on a template parameter when the template parameter is used as abase class, but I think there are a couple of cases that require clarification.

First, consider a template parameter as an *indirect* base class:

```
struct A {
      struct B { /*...*/ };
      int a;
      int Y;
};
template <class T> struct AA : public T {};
int a;
template <class T> struct Y : public AA<T> {
      struct B { /*...*/ };
      B b; // B defined in Y ??
      void f(int i) { a = i; } // ::a ??
      Y* p; // Y<T> ??
}
Y<A> ya;
```

In short, can the conclusions of 14.6.2/5 still be drawn when the template parameter is an indirect base class?

The second example is a bit more complex and has to do with template partial specializations:

```
template <class T> struct A {};
template <> struct A<int> {
      struct B { /*...*/ };
      int a;
      int Y;
};
int a;
template <class T> struct Y : public A<T> {
      struct B { /*...*/ };
      B b; // B defined in Y ??
      void f(int i) { a = i; } // ::a ??
      Y* p; // Y<T> ??
}
Y<int> ya;
```

In short, can the conclusions of 14.6.2/5 still be drawn when the template parameter is involved in the selection of a base class which is a template specialization (or partial specialization for that matter)?

The case involving a template specialization as a base class is more difficult, because on one hand you want declarations of a "normal" base class to override declarations where appropriate, but you don't want members of indeterminate base classes to override.

I suggest that the language of 14.6.2/5 be amended to read something like:

"If a base class of a class template is one of:

- a template-argument.
- a template-id whose argument list contains a template-argument, where the correct specialization of template-id cannot be chosen until the template-argument is known.
- a template-id whose argument list contains a template-argument, and which has a base class falling into one of the two cases above.

then a member of that base class cannot hide a name declared with a template, or a name from the template's enclosing scopes."

respectfully submitted,
John Lilley
jlilley@empathy.com
Nerds for Hire, Inc.
4270 Evans Dr.
Boulder, CO 80303

RESPONSE: Rejected. Already clear; 14.6.2(temp.dep) paragraphs 3 and 4 also apply to indirect base classes; base classes and their members are not considered if they depend on the template parameter even if the base class is explicit specialized.

<<<<<< Public Comment #09/Owen" follows >>>>>>
Date: January 27, 1997

I would like to request that the "string" class be fully integrated into the C++ standard libraries as a total replacement for null-terminated character arrays.

For instance all functions that take char* as input should take a string, instead, such as file name inputs for file stream classes. This is probably obvious and is fully backwards compatible.

But in addition, I feel that almost all functions that presently use a char* argument for output should use a reference to a string, instead. This includes the getline() and str() methods associated with all iostream classes. This will not be backwards compatible, but this seems a small price to pay for the gain in robustness and self-consistency we would gain.

Then one could eliminate the "getline" global function, which seems to be a hack to work around the lack of integration of "string" into the standard libraries.

Presently there seems to be no obvious and simple way to extract all remaining data from a stringstream into a string. If true, this is a very serious oversight and one I hope will be addressed. The solution I suggest is to allow getline to accept an "int" for the terminating character (presently it requires a "char"). Then one could use getline(..., traits::eof()) to extract the data.

Thank you for your consideration,

Russell E. Owen
owen@astro.washington.edu

RESPONSE:
Paragraphs 1-4 previously considered and rejected.
Paragraph 5 considered and closed with no action taken.

<<<<<< Public Comment #10/Owen" follows >>>>>>
Date: January 27, 1997

Please consider providing a C++ flavor of "assert" that throws an exception.

I realize that writing such a thing manually is possible, but making such important core tools standard has obvious benefits.

I can see several solutions:

- Simply change the existing "assert" so that it throws an exception. This would simplify the programmer's life for new code, as there would only be one way to do this obvious and useful thing. However, it may cause trouble for existing code.
- Provide a new function with a name similar to "assert", but not identical. This would be safest for existing code, but adds clutter.
- A compromise wherein a #define determines the behavior. This is my least favorite solution, though it is still arguably an improvement.

Russell E. Owen
owen@astro.washington.edu

RESPONSE: Previously considered and rejected.

<<<<<< Public Comment #11/Owen" follows >>>>>>
Date: January 27, 1997

I am writing to beg the C++ committee to require STL to have bounds checking and generally be robust against programmer error.

If necessary for performance issues, it should be possible to disable the bounds checking (possibly by setting a compiler option, pragma or #define; it need not be trivial). But the default should have bounds checking ON. C and C++ programmers and the users of their code have suffered enough in the cause of performance.

Thanks for your consideration, on what is possibly a tender topic.

Russell E. Owen
owen@astro.washington.edu

RESPONSE: Previously considered and rejected.

<<<<<< Public Comment #12/Girod" follows >>>>>>
Date: February 6, 1997

Hello!

Please find some comments on the committee draft for the C++ language standard.

I sent them as well to the Australian standard body, which forwarded at least part of them to the editor <c++std-edit@research.att.com>.

My coordinates are to be found in my "signature" below.

Best Regards
Marc Girod

----
* In [temp.friend] (14.5.3 (4)), 2nd line: remove the word "function"

```
  template  class.   In  this case, the corresponding member function of
                                                              ^^^^^^^^
```

  The "member of a class template" is not necessariliy a member "function". In the example, one member is a nested class.

* In [temp.friend] (14.5.3 (4)): add an example for a typedef member, using the typename keyword.

* In [dcl.type.elab] (7.1.5.3 (3)): add one line (and optionally the example)

```
          friend typename identifier ;
  [Example:
          template <class T> class Y {
              typedef T::Session S;
              friend typename S;
          };
  ]
```

* In [dcl.type.elab] (7.1.5.3 (4)): add one line (and optionally the example)

```
          friend typename nested-name-specifier identifier ;
  [Example:
          template <class T> class Y {
              friend typename T::Session;
          };
  ]
```

  Rationale: these are implicitly made legal by [temp.friend], since nested typedefs are class "members". It raises thus an ambiguity that these cases are missing from the lists of explicitly allowed ones.

* A related issue: in [temp.explicit] (14.7.2 (2)), it is not clear whether the use of a typedef is allowed or not. I suggest to explicitly allow it, and to provide an example such as:

```
          typedef deque<Callback*, allocator> dCa;
          template class dCa;
```

* Also in [temp.explicit] (14.7.2 (2)), there is no syntax for explicitly instantiating specific members of a template class. This means that in some cases, implicit instantiation may succeed, where explicit instantiation is impossible...

  The syntax could be:

```
  [Example:
          template void deque<Callback*,
                  allocator>::deallocate_at_begin();
  ]
```

--
Marc Girod Kilo RD 4 Phone: +358-9-511 27703
Nokia Telecommunications P.O. Box 370 Fax: +358-9-511 27432
00045 NOKIA Group Finland marc.girod@ntc.nokia.com

RESPONSE:
1) accepted.
2) rejected -- you cannot declare a typedef member to be a template.
3) rejected -- the suggested syntax is not allowed.
4) rejected -- the suggested syntax is not allowed.
5) rejected -- the suggested syntax is not allowed.
6) accepted -- 14.7.2(temp.explicit) para 2 shows how this can be done.

<<<<<< Public Comment #13/Brasfield" follows >>>>>>
Date: February 21, 1997

In the December 1996 draft C++ standard, in section 13 Overloading, the following text can be found at 13.3.1.2.9:

```
  13.3.1.2  Operators in expressions                   [over.match.oper]
. . .
9 If the operator is the operator ,, the unary operator &, or the opera-
  tor  ->, and overload resolution is unsuccessful, then the operator is
  assumed to be the  built-in  operator  and  interpreted  according  to
  clause _expr_.
```

As I read the above, I wonder if "unsuccessful" is what was meant. Earlier, (at 13.3[4]) the phrase "overload resolution succeeds" is said to occur when a best viable overload exists and is unique, (resulting from finding the best viable overload on each argument).

This seems to defeat the purpose of generally making ambiguity an error in C++. Under the usual regime, where ambiguity is an error, if I add an overload that creates an (unintended!) potential ambiguity, and I use it in a way that creates an actual ambiguity, the result is an error and I have to decide what I really meant to occur. But with the above rule, where the overload reverts to the built-in operator upon "unsuccess", which can result from adding an overload that gives rise to an ambiguity, I can get a silent change in behavior due to the ambiguity.

I suggest that the phrase "and overload resolution is unsuccessful" of section 13.3.1.2.9 should be "and no viable overloads can be found for the operator". At the very least, the meaning of the word "succeeds" in this context needs to be qualified.

--Larry Brasfield
larrybr@earthlink.net
Above views are mine alone.

RESPONSE: accepted -- 13.3.1.2(over.match.oper) paragraph 9 was clarified.

<<<<<< Public Comment #14/Finney" follows >>>>>>
February 22, 1997

Page 2_3. Paragraph (3) says "Each ? that does not begin one of the trigraphs listed above is not changed." but paragraph (4) says that "?????????" becomes "???" which implies that ??? is a trigraph and is replaced by ?, but that contradicts paragraph (3).

Michael Lee Finney
114 Old Wiggington Road
Lynchburg, Va. 24502-4669
804/385-4468
mfinney@lynchburg.net

RESPONSE: accepted -- the example in 2.3(lex.trigraph) paragraph 4 was removed.

<<<<<< Public Comment #15/Horwat" follows >>>>>>
February 24, 1997

I'd like to submit a short comment to the 2 Dec 1996 draft of the C++ programming language (ISO/IEC 2nd CD 14882).

The issue is the interaction of template instantiation and partially defined classes. Consider the following example:

```
#include <list.h>

struct S {
      int a;
      list<S> b;
      };
```

Is this meant to be legal C++? The answer depends on whether the expansion of the list template tries to allocate a field of type S in the class list<S>. If so, it would violate paragraph 9.2.8 which states that non-static members of a class must be objects of previously defined classes. However, I couldn't find anything in the draft standard that states that list<S> may or may not expand into a class with a field of type S.

Please specify the behavior of definitions of all container templates (list, vector, etc.) in the standard library with respect to template parameters that are partially defined.

Dr. Waldemar Horwat
individual
976-1 Alpine Ter.
Sunnyvale, CA 94086
408-749-9708
waldemar@acm.org

RESPONSE: accepted -- a template argument may be an incomplete class type.

<<<<<< Public Comment #16/Aldridge" follows >>>>>>
February 25, 1997

Name: John Aldridge
Company: Graphic Data Systems

Address: Wellington House
East Road
Cambridge
CB1 1BH
ENGLAND

Phone: +44 1223 371925
E-mail: jpsa@uk.gdscorp.com

I cannot find wording in the draft which unambiguously says whether the following example should compile:

```
class A {
public:
   void B ();
private:
   enum X {X1, X2, X3};
};

void A::B ()
{
   struct Z {X x; int i;};
}
```

Section 11.8 (Nested classes) says:

> The members of a nested class have no special access to members of an enclosing class ...

but I cannot find an equivalent statement about the access rights of local classes.

RESPONSE: accepted -- 9.8(class.local) paragraph 1 now addresses this question.

<<<<<< Public Comment #17/Bau" follows >>>>>>
February 25, 1997

Paragraph 2.3.1 + 2.3.4, Trigraph Sequences:

2.3.1 contains a list of trigraph sequences that should be replaced by single characters; 2.3.4 contains further rules and an example explaining these rules.

The example in 2.3.4 does not make any sense. It only makes sense if I assume that the sequence ??? should be replaced by a single question mark ?. However, the sequence ??? is not mentioned in 2.3.1.

Either 2.3.1 must be changed to include that the sequence ??? is replaced by ?, or the example in 2.3.4 must be changed.

Christian Bau
Insignia Solutions Ltd.
EMail: christian.bau@isltd.insignia.com

RESPONSE: accepted -- the example in 2.3(lex.trigraph) paragraph 4 was removed.

<<<<<< Public Comment #18/Ward" follows >>>>>>
February 25, 1997

Requestor: Judy Ward
Company: Digital Equipment Corporation
Address: ZK02-03/N30
           110 Spitbrook Road
           Nashua, NH 03062-2642
           USA
Telephone: 603-881-2687
Email: j_ward@decc.enet.dec.com

ANSI C++ Public Review Comment:

The standard fstream classes are missing a key feature that most existing fstream classes have, namely the ability for users to access the association between an streambuf and the underlying C file descriptor/pointer.

For example, most iostream classes have these member functions to create or attach a C file to a C++ fstream:

```
filebuf::filebuf(int file_descriptor,...)
filebuf* filebuf::attach(int file_descriptor, ...);
```

Most existing iostream classes have a way to access the underlying C file descriptor or pointer given the fstream, i.e:

```
      int filebuf::fd() const;
```

We think this functionality is essential for C++ users who need to work with other C library features, i.e. sockets, extensions to stdio for special file types, etc.

We understand that file descriptors are not in the C standard, but C FILE* pointers are included. So changing the above functions to accept or return C FILE pointers would be fine.

We also understand that this would require implementors to use the underlying C input/output libraries to implement iostreams. We don't know of any vendor who does not plan to do that anyway.

Alternatively, one could consider writing a stdiobuf class to encapsulate these conversion functions (to connect a streambuf to a FILE*), but it's probably too late for that.

RESPONSE: Became US issue 27-039. Was considered too big a change at this point, and so was rejected for now.

<<<<<< Public Comment #19/Whitman & Ward" follows >>>>>>
February 25, 1997

Public Review Comment ISO/IEC 2nd CD 14882,C++ Language:
C library names should be removed from namespace std

We believe that C library names should be removed from namespace std. The draft currently states (Clause 17, Annex D) that the C++ Standard library will provide 18 ISO C library headers in a <cname> form which brings ISO C names into the namespace std and a <name.h> form which bring ISO C names into both the std and global namespace (excluding macros).

We believe that the implementation for this is highly error prone, leading to unmaintainable C headers and serious bugs. Some of our major concerns are:

- o maintaining duplicate copies of the .h headers, one supplied by C and one by C++.
- o adding complex macros to headers to avoid nested namespaces.
- o ensuring that names are consistently available (or not) in namespace std regardless of the order of header file inclusion in a user program.
- o coordinating an effort to modify, rewrite, reorganize C headers supplied by a C development environment which is outside of the scope of the C++ environment.

We believe that in practice the benefits of putting ISO C names into namespace std do not outweigh the increased complexity required for compliance. The burden of this support is not limited to C++ compiler/library vendors. It will impact any independent C++ library/tool vendor and operating system provider all of which will need to ensure that the correct C/C++ header interfaces are in place.

This was discussed in depth on the library reflector. For details see messages 4598-4611,4614-4615,4618-4626,4628,4630,4632-4636, 4638-4641,4643,4645-4647,4650-4656,4662-4664,4666,4676,4689,4690

The resolution is to change the Working Paper as follows:

- o 17.3.1.2 table 12, C++ Headers for C Library Facilities delete the leading "c" from header names and append ".h".
- o Remove 17.3.1.2 paragraph 4, 7 and footnote 153. Add the ".h" headers place all their names into the global namespace.
- o Delete from Annex D the [.depr.c.headers] section.
- o Change references to std::ISO-C-name to ISO-C-name

Requestor: Sandra Whitman, Judy Ward
Company: Digital Equipment Corporation
Address: ZK02-03/N30
110 Spitbrook Road

Nashua, NH 03062-2642
USA
Telephone: 603-881-2687
Email: whitman@tle.enet.dec.com
j_ward@decc.enet.dec.com

RESPONSE: Became US issue 17-004; some WP changes were accepted.

<<<<<< Public Comment #20/Sachs" follows >>>>>>
February 25, 1997

Comments about 1996 C++ Draft standard ISO/IEC JTC1/SC22

David Sachs
1069 Rainwood Drive
Aurora, Illinois 60506-1351
Email: sachs@fnal.fnal.gov

to: X3 Secretariat
Attn.: Deborah J. Donovan,
1250 Eye Street, NW,
Suite 200,
Washington, DC 20005,
Email: ddonovan@itic.nw.dc.us.
cc: ANSI
Attn.: BSR Center
1 West 42nd Street
New York, NY 10036
Email: wluk@ansi.org

These comments will be sent both by Email and via the U. S. postal system.

1) Trigraphs

I do not understand the trigraph examples in section 2.3, page 2-3 of the draft standard. These examples act as if there were a 10th trigraph sequence ("???" replaced by "?") in addition to the 9 listed sequences. However the draft standard explicitly denies that there are any unlisted sequences,

The example in question, Section 2.3 paragraph 4 reads:

[Example: The sequence "???=" becomes "?=" not "?#". The sequence "?????????" becomes "???", not "?". -- end example]

2) Direct and indirect copies of same base class

Paragraph 3 of section 10.1 (page 10-2) of the draft standard specifically declares the following construct to be well-formed (irrelevant lines omitted):

```
class L { public int next; /*   */ };
class A : public L { /*    */  };
```

```
class D : public A, public L { void f();  /*   */ }; // well-formed
```

The class D is presented as if it were a perfectly normal, usable class. The only restriction on such a class anywhere in the draft standard is in section 12.6.2 paragraph 2 (page 12-13), which disallows a mem-initializer for the duplicated base class if there is an indirect virtual copy and the direct copy is not virtual.

In fact, unless the standard is changed such as by allowing a direct base class to hide an indirect copy of the same base, any attempt to access the direct copy of the duplicated base class or its members, except by methods best suited for use in obfuscated C++ contests, is ambiguous and causes a compile time error.

As a minimum the text of the standard should warn of the very limited usability of such a class.

3) Recursive exceptions

I can find nothing in chapter 15 of the draft C++ standard either permitting or prohibiting recursive exceptions.

I use the term "recursive exception" for the situation described in the following paragraphs:

After an exception is thrown, the runtime stack is unwound while searching for a proper handler. As part of the unwinding process destructors are called for class objects in the stack.

Suppose that a destructor called in this manner, or a function called from such a destructor contains a try block. If something within the scope of such a try block throws an exception, then at that point, there are 2 uncaught exceptions being processed.

I call the second exception a "recursive exception". There probably is a better term for this.

Obviously, the second exception must be caught and processed by an exception handler within the destructor or a function it calls; the standard properly specifies that having the destructor terminate by throwing an exception requires terminate() to be called.

The draft standard does not specify whether a properly caught recursive exception is standard-conforming.

Messages in the comp.std.c++ newsgroup indicate that it is apparently the intent of the standards committee that standard conforming compilers are required to support recursive exceptions as described above.

4) Ambiguous mem-initializer-id

The draft standard section 12.6.2 paragraph 2 declares that a mem-intializer that is ambiguous because its mem-initializer-id designates both a direct non-virtual base and an inherited virtual base class is ill-formed.

There is no similar statement for a mem-initializer-id, that designates both an initializable base and a class member. If such a mem-initialer is not ill formed, what does it initialize?

5) Throwing an exception example

The example in section 15.1 paragraph 1 needs to be changed slightly because of the recent change of the type of string literals from char* to const char*. It currently reads:

... throw "Help!"; can be caught by a handler of some char* type ...

char* probably should be changed to const char*. The sample code following this sentence does properly use const char*.

David Sachs - Fermilab, MSSG MS369 - P. O. Box 500 - Batavia, IL 60510
Voice: 1 630 840 3942 Department Fax: 1 630 840 3785

***RESPONSE*:**
1) accepted -- the example in 2.3(lex.trigraph) paragraph 4 was removed.
2) accepted -- see clarifications in 10.1(class.mi) paragraph 3.
3) accepted -- see 15.2(except.ctor) paragraph 3.
4) accepted -- see 12.6.2(class.base.init) paragraph 2.
5) accepted.

<<<<<< Public Comment #21/Dimm" follows >>>>>>
February 25, 1997

From:
Bill Dimm
275 Bryn Mawr Ave. Apt. M14
Bryn Mawr, PA 19010
(610)995-1570
billd@gim.net or billd@mop.com
Employer: BNP/Cooper Neff, Inc.

1) p. 2-6, Table 3 - Missing "export"
The table of keywords is missing the "export" keyword described on page 14-1.

2) section 21.3 - Add basic_string::push_back
The basic_string class should have a push_back member function (as in Table 69, p. 23-5) to make it more compatible with the other container classes.

3) p. 23-23 and p. 23-25 - vector::resize Pass by Value?
The second argument for vector::resize is passed by value (instead of reference to const object). In the absence of a compelling reason for pass by value, this should be changed to use a reference to const (for greater efficiency and more uniformity in the library). Page 23-25 defines vector::resize in terms of vector::insert (which uses a reference), so it is surprising to see that the two functions treat their arguments differently.

4) section 23.2.4.3 - vector::insert under specified
The draft makes no statement about whether or not pointers/references remain valid DURING (not after) vector::insert. Since the value being inserted is a reference to const object, it is unclear whether or not you can insert an element of a vector into another location of that vector. For example (recalling from p. 23-5 Table 69 that push_back is defined in terms of insert):

```
   vector<int> v(100);
   v.push_back(v[0]);  // is this well defined?
```

The library implementations that I have seen do accommodate the code above because (when capacity must be increased) they fill-in the new memory region completely before destroying the objects in the original memory. I would suggest that the committee require that references into the vector remain valid during (but not after) the insertion. If such a restriction is not imposed, I would suggest that the standard explicitly say that code like the example above is undefined.

RESPONSE:
1) Accepted.
2) Became issue 21-006; considered and closed with no action taken.
3) Became issue 23-008; considered and closed with no action taken.
4) Became US issue 23-009; a proposed resolution was accepted.

<<<<<< Public Comment #22/Morse" follows >>>>>>
February 25, 1997

The following comments are being submitted by email with follow up to the X3 Secretariat and a copy to ANSI during the public review period for ISO/IEC CD 14882 (X3J16).

NOTE: These comments are based on the 24 September 1996 draft and thus may vary from the final draft.

ITEM 1:

Section 16.1 Paragraph 4 states "The resulting tokens comprise the controlling constant expression which is evaluated according to the rules of 5.19 ..."
Section 5.19 deals with general constant expressions in the language including several features which are not usable in constant expressions for #if. Among these features are: sizeof, casts, enumeration types, and floating types. Further, paragraph 1 of section 5.19 contains several forward references to other paragraphs but does not contain a reference to section 16.1.

Recommendations:
1. Add a forward reference to paragraph 16.1 in section 5.19 paragraph 1.
2. Add language to section 16.1 paragraph 4 which enumerates those items of section 5.19 which do not apply (e.g. casts, sizeof, etc). Alternatively, add language to section 5.19 to accomplish the same objective.

ITEM 2:

Section 16.3.2 paragraph 1 first sentence states "a parameter is immediately preceded by a # preprocessing token". It is my experience with current compilers that is it acceptable to have white space tokens between the # and the parameter.

Recommendations:
Add language to section 16.3.2 to indicate that white space can appear between the # and parameter if this is the intent of the standard.

ITEM 3:

Section 16.3.4 deals with rescanning and further replacement. My literal reading of this section does not seem to address one of the examples given in section 16.3.5 paragraph 5. The latter portion of the first example includes the string "% t(t(g)(0) + t)(1)" resulting in the string "% f(2 * (0)) +t(1)". The parameter of the first t macro expands to "f(2 * (0) + t)".

A literal reading of section 16.3.4 would suggest that this expanded parameter should be inserted, rescanned and expanded again resulting in "f(2*(2*(0) + t))(1)". Section 16.3.4 does suggest that nested macros are not expanded, however, the current wording does not seem to cover the case where the macro was expanded in the parameter expansion. Section 16.3.1 suggests that parameters are fully expanded and substituted while 16.3.4 covers rescanning without any suggestion of an interaction between the two.

Recommendations:
Section 16.3.4 paragraph 2 sentence 2 reads "Further, if any nested replacements encounter the name of the macro being replaced, it is not replaced." After this sentence I suggest adding: "Any nested replacements encountered during parameter expansion continue to be unavailable for further expansion after parameter substitution and subsequent rescanning."

Submitted By:

Peter L Morse
177 Telegraph Rd #501
Bellingham, WA 98226
Phone: (520) 574-5446 or (206) 952-0494
Email: MorseRover@aol.com

***RESPONSE*:**
Item 1) & Item 2) rejected -- the committee has chosen to keep Clause 16 on the preprocessor identical to the preprocessor section in the C standard.
Item 3) rejected -- request for an extension.

<<<<<< Public Comment #23/Parker" follows >>>>>>
February 26, 1997

Name: Brian Parker
Business Name: Voxel Graphics Research
Email: bparker@gil.com.au
Postal Address:
14 Jules Av,
Rochedale South,

Brisbane QLD 4123,
Australia

Phone: +61 7 32992807

Comments by section follow.

---

(1) Page 14-22 Sec 14.6.2 Clause 2 has an example

```
template<class T> class Z {
public:
      void f() const {
         g(1);
      }
};
void g(int);
```

and states that this is ill-formed because g(1) is not type-dependent and so is looked up at template definition. This is confirmed in an example in Sec 14.6.3. However, the example goes on to state that if Z had been instantiated with an int, then g(1) would become type-dependent and hence the example would be well-formed. My understanding was that g(T(1)) would be required to make g(1) type-dependent. Earlier in the example, three ways a function call can be type-dependent are listed and the second seems to imply that g(1) is type-dependent for T an int. The third is a (non-template) example that in a template class would seem to me *not* to be type-dependent.

---

(2) Page 14-26 Sec 14.6.5 Clause 2 and footnote 126 states that a friend declaration in a template class does not inject the name into any scope. It then gives an example of

```
a = gcd(a,b)
```

and states that gcd is looked up inside number<double> by the argument-dependent lookup rules in 3.4.2. By my reading, however, 3.4.2 states that lookup starts in the enclosing namespace, not inside the template class.

---

(3) Page 14.30 Sec 14.7.2 gives examples with two syntaxes for an explicit instantiation of a template function where all the arguments can be deduced i.e.

```
template void sort(Array<char>&);
```

or

```
template void sort<>(Array<char>&);
```

Are these equivalent by design?
This same issue arises in explicit specialization Sec 14.7.3

---

(4) Page 5-23 Sec 5.9 Clause 2 describes how comparing two (non-null/ non-void) pointers converts them to a composite pointer type similar to one of them, but doesn't state which one. I think this should state that it is the pointer that has an implicit conversion from the other one (if it exists).

---

(5) Page 3-20 Sec 3.5 Clause 6
In the example, why does extern int i have external linkage and not the internal linkage of the earlier static i definition?

---

(6) Page 18-15 Sec 18.5.1 Clause 7 defines type_info::name() as implementation-defined, so a conforming implementation could simply return a null string for all types, effectively making name() unusable. Ideally, it should be defined to return the type name in some canonical form e.g. a fully-qualified elaborated type-id with no redundant spaces (although e.g. pointer non-type template parameters would require further specification). This would allow name() to be used to label types in a persistence library (e.g. a recent Microsoft Systems Journal described such a library). Failing this, I think that at least name() should be defined to return a unique string for each type to allow type_info to be used as a hook to further user-defined type information (as envisaged by Dr Stroustrup in D&E.) In D&E page 318, it is suggested that typeid(*p).name() or &typeid(*p) could be used as an index into a map for this purpose, but currently neither expression is defined to be unique for different types.

---

(7) Page 18-14 Sec 18.5.1
A significant limitation of the current draft C++ is that given a pointer (or reference) to some type or derived type, it is impossible to make a copy of the most-derived object. e.g.

```
template<class Allocator>
class Myclass{
public:
	const Allocator* local_copy;

	Myclass(const Allocator& alloc = Allocator())
		// The passed in alloc may be a default temporary or a user allocated
		// derived object. Make a local copy in either case.
	: local_copy(clone_ptr(&alloc))	// Oops, no such clone_ptr function exists
	{}

	~Myclass() {delete local_copy;}
	...
};
```

Therefore, one can only make local copies for types that have explicitly added clone() virtual functions to the base-class (and maintained them in all derived classes).

If, however, class std::type_info was extended with the member function clone as follows-

```
class type_info {

	... rest as per the draft standard

	virtual void* clone(const void* const p) const = 0;
};
```

where clone() is overridden for the type that the type_info represents such that clone() copy-constructs a copy of p on the heap

e.g. for type_info representing type T, this would be overridden by the implementation as...

```
T* type_info::clone(const void* const p) const
{
	return new T(*(const T* const)p);	// undefined behaviour if p not a T*
}
```

(If the copy constructor of the object is inaccessible a function that returns a null pointer would be generated instead).

Given the above function, one can now define the following (in std namespace) type-safe template function-

```
// return new heap copy of object pointed to by ptr
template<class T>
T* clone_ptr(const T* const ptr)
{
      return typeid(*ptr).clone(ptr);
}
```

This new member function of type_info would be easy to implement given the current type_info implementation and shouldn't add any code size overhead (the function would only be generated if used).

In fact, just clone_ptr() could be defined in the standard and the definition of the member function in the type_info class could be left as an implementation detail.

Another advantage of this function is that it allows a value semantics, polymorphic smart pointer template to be written that works on any (copyable) type which would simplify the design of classes using dynamic memory allocation.

(Note 1: I have a comment about covariant return types. As written, the declaration of type_info::clone() is incorrect according to the draft std as the covariant return type must be derived from the return type of the base class, but in the spirit of void* being viewed as the base of all pointers (i.e. all pointers can be implicitly cast to void*) might it not be more consistent to allow derived class return types of any pointer type to override void* returns in the base class? In any case, this doesn't impact the above function; just define the function in derived classes to return void* and later (safely) cast the returned void*.)

(Note 2: If a type_info member that gave the sizeof the type was available, then type_info::clone could alternately be written using placement new and clone_ptr() could actually allocate the memory. This would be a more flexible scheme.)

----

(8) Page 20-5 Sec 20.2.1
Rather than putting the relational operators in a nested namespace "rel_ops" it may be better to put them in a class-

```
template<typename T>
class rel_ops {
      friend bool operator!=(const T& lhs, const T& rhs){
            return !(lhs == rhs);
      }
      ... and so on for >, >= & <=
};
```

Then the names could be injected for the required type where required by explicit template instantiation-

```
template rel_ops<Mytype>;
```

or by deriving from rel_ops<T>

----

(9) Page 14-34 Sec 14.7.3 Clause 16
The example isn't of a member template as stated.
----

(10) Page 18-10 Sec 18.4
Throughout this section, various placement delete functions are described as being called by a delete-expression. My understanding was that the placement delete functions were only called if an exception was thrown during a new expression. When are the "nothrow" placement delete functions called. Page 18-13 Sec 18.4.1.3 Clause 8 states that operator delete(void* ptr, void*) is the "default function called for a placement delete expression". What is a placement delete expression?
----

(11) Page 18-17 Sec 18.6.2.2 Clause 2
Section 15.5.2 states that the unexpected() function can throw any exception and those not in the function's exception specification will be converted to bad_exception if that is in the exception specification. This section, however, states that a user supplied unexpected_handler must not throw exceptions not on the exception specification. What is the reason for this restriction?
----

(12) Various trivial editorial changes:
Grammar page A-5: There are two identical id-expression productions.
Page 5.79 Sec 5.2.7 Clause 9: last two dynamic casts in void g() should be from &d not &dr.
Page 14-26 Sec 14.6.4.2: "not just considered" to "not just considering".
Page 14-34 Sec 14.7.3 Clause 16: "specialized class is not be" to "specialized class is not".
Page 14.34 Sec 14.8 Clause 2 "Each function template" to "each function template specialization".
Page 14-41 footnote 128: "non-teplate" to "non-template"
Page 13-24 Sec13.6 Clause 15: There is no footnote 123.
Page 17-8 Sec 17.3.1.3 Clause 2: "implementation has has" to "implementation has"
Page 23-20 Top of page "nmespace" to "namespace"

RESPONSE:
1) accepted -- the example was removed.

2) accepted -- 3.4.2(basic.lookup.koenig) now discusses associated classes and namespaces.

3) rejected -- yes, they are equivalent.

4) rejected -- the committee believes the WP is sufficiently clear.

5) rejected -- because the static *int i* in global scope is hidden. The text in 3.5(basic.link) paragraph 6 explains this.

6) Page 18-15 Sec 18.5.1 Clause 7) Considered, then closed with no action taken.

7) Page 18-14 Sec 18.5.1) Rejected as previously considered.

8) Page 20-5 Sec 20.2.1) Rejected as previously considered.

9) accepted -- see 14.7.3(temp.expl.spec) paragraph 5 and 6.

10) Editorial. The text in the library section 18.4 is incorrect and needs to be changed to match the rules for placement delete in 5.3.4.

11) Editorial. The text in the library section 18.6.2.2 must be changed to match the rules in 15.5.2.

12) Proposed changes were accepted.

<<<<<< Public Comment #24/Moore" follows >>>>>>
February 27, 1997

Comments from:

David L Moore
Advantest America R & D
3201 Scott Boulevard
Santa Clara CA 95054

(408) 727 2222 x386

d.moore@advantest.com

The following are my personal remarks concerning the Draft C++ standard. Although these problems have mostly been discovered during the course of my work, they do not represent a position of my employer.

Part A deals with what appears to be a serious problem.

Part B deals with points that I believe could be clarified. The lack of clarity may arise from my lack of familiarity with the standard rather than actual problems. Alternatively it may be felt that the possibility of divergent implementations caused by these difficulties is small.

Possibly, these points should be addressed in an annotation to the standard in a similar vein to that for Ada if the benefits of clarification are felt to be outweighed by the delay that would be caused.

Part A. Serious Problems.

1/ Exceptions.

The current language appears to not correctly specify the order of searching for an exception handler. Consider the following code fragment:

```
  try {
      throw 2;
      }
  catch (int i) {
      throw 2.0;
      }
  catch (double a) {
      cout << "what am I doing here";
      }
```

According to the draft standard, this will print "what am I doing here". I believe that this is not the desired semantics. Indeed, these new semantics makes rethrowing exceptions impossible.

The reasoning for my claim is as follows:

i) Note that this entire construct is a try block. From 15, para 1:

```
try-block:  try compound_statement handler-seq
```

ii) From 15.1[2]

When an exception is thrown, control is thrown to ... the handler whose try block was most recently entered by the thread of control and not yet exited.

We have not yet exited the above try block as, according to the syntax, the handlers are part of the try block.

SUGGESTED FIX:

change the grammar to:

```
try-block:  try-protected-statements handler-seq
try-protected-statements: try compound-statement
```

and replace the words "try block" in 15.1 (2) by "try protected statements".

Part B. Clarifications Desirable.

1/ What happens when a destructor throws an exception while we are unwinding the stack in preparation for entering a handler?

Suppose that in the process of unwinding the stack in preparation for entering a handler[15.2 para 1], an exception is thrown. What are the semantics?

2/ Should it be possible to catch a throw of '0' with a handler that catches void *?

```
try {
    throw 0;
    }
catch (void *) {
    cout << "At least some compilers print this\n";
    }
```

Many compilers allow this. The section 15.3(3) references 4.10 which discusses the fact that 0 can be converted to a pointer type. However, as it is not of a pointer type, the language of 15.3(3) :

the handler is of type cv1 T* <u>cv2 and E is a pointer type</u>...

appears to prohibit this behaviour. This case should be clarified.

3/ The "unexpected" procedure.

Can this be declared static:

```
static void unexpected();
```

If not, is it an error to declare such a function? The implication of the manual appears to be that this cannot be static, but it possible that this is wishful thinking on my part as an implementor.

BTW 18.6.2 is empty. Is this intentional?

4/ bad_alloc.

5.3.14 para 16:

> The allocation function can indicate failure ...

Does can here mean "may" or does it mean "shall"? One of these words should be substituted for "can". There is at least one other instance where can is used when may or shall should have been used.

5/ Elision of temporaries.

At various points, the statement is made that temporaries can be removed when removing them causes no semantic changes "except for calling constructors and destructors".

Does this mean that any code in those constructors and destructors can be ignored when deciding that the temporary need not be created. For example, can a temporary for the following class fragment always be deleted:

```
class X
  {
  public:
    static int i;
    X(X& x) {i++;}
    ~X()    {i++;}
```

note that not creating and destroying a temporary reduces i by 2.

(I believe this should be the case since, in my view, the above code should not be considered well formed)

***RESPONSE*:**

Part A) accepted and fixed.
Part B1) accepted -- see 15.2(except.handle) paragrap 3.
Part B2) accepted -- a handler of pointer type is not considered.
Part B3) section [lib.using.linkage] requires all library functions to have external linkage.
Part B4) accepted -- see 5.3.4(expr.new) paragraph 16.
Part B5) no action -- yes, given the example, temporaries of type X can be eliminated.

<<<<<< Public Comment #25/Parker" follows >>>>>>
February 27, 1997

Name: Brian Parker
Business Name: Voxel Graphics Research
Email: bparker@gil.com.au
Postal Address:
14 Jules Av,
Rochedale South,
Brisbane QLD 4123,
Australia

Phone: +61 7 32992807

Comments by section follow.

Page 24-23 Sec 24.5.3.2 Clause 3
Earlier it is stated that class proxy was for exposition only and need not be supplied, but here a constructor taking it is required.

Page 23-22 Sec 23.2.4 [lib.vector]
From the definition of vector<T> given, it appears that an implementation can only ever grow a vector and never reclaim its storage. At the least, a user of the class can not assume otherwise. For example,

```
void f()
{
	vector<int> v1(1000000), v2(1000000); // initially large vectors
	vector<int> v3(1);	// initially small vector

	int* pi = &v1[1];
	v1.erase(v1.begin()+1, v1.end());
	// v1.capacity() >= 1000000, v1.size() == 1 here as the standard (necessarily)
	// specifies that storage is not reallocated so *pi remains valid

	... other operations on v1

	// at this point we know there are no iterators or references that need to remain
	// valid so we would like to reclaim storage on v1
	v1.reserve(1); // This won't work, reserve() is defined to only increase
	               // capacity() not decrease it.

	v1.resize(1);  // This won't work, resize() is defined in terms of erase() which
	               // is defined not to reallocate.

	v1.compact();  // Ideally, an operation like this would be defined such that
	               // (for the default allocator at least) v1.capacity() will be
	               // equal to vector<int>(v1.size()).capacity() i.e. the same storage
	               // overhead as a newly initialized vector of the same size.

	v1.clear()     // One would expect that this would free all allocated memory
	               // but clear() is defined in terms of erase() and so this is not
	               // guaranteed. clear() should be defined such that
	               // v1.clear().capacity()==vector<int>().capacity() is postcondition
	               // (at least for default allocator) i.e. it has only the same
	               // storage overhead as a newly initialised default empty vector.

	v2 = v3;       // One would expect that v2 now only allocates as much memory
	               // as required, but this is not guaranteed by the standard. It
	               // could still have v2.capacity() >= 1000000. Ideally assignment
	               // would be specified for vector such that
	               // (v2 = v3).capacity() == vector<int>(v2.size()).capacity() is
	               // a post-condition .
}
```

The post-conditions discussed above should be easy to guarantee for the default allocator- the allocator storage overhead is implementation-specific but deterministic. At the minimum, they could degrade to suggested behaviour for user-supplied allocators.

In summary, I think that at least clear() should be defined with the post-condition above to give the user some explicit control over memory leakage. Preferably, the function compact() would be added (or v1.resize(v1.size()) defined to do the same) and the assignment behaviour specified as above. The container classes' definitions are careful to give time complexity guarantees without which they would not be usable in many situations. I think that some minimum guarantees on space complexity are also required.

Note: these comments also apply to basic_string.

Various trivial editorial changes:
Page 21-4 Sec 21.1.3 Clause 8 "derived classed" to "derived classes"
Page 23-6 Sec 23.1.2 Clause 4 "equal keys" to "equivalent keys"
Page 23-38 Sec 23.3.4 Clause 2 "the a_eu operations" to "the a_eq operations"
Page 24-20 Sec 24.5.1.1 Clause 3 "a copy of s" to "a copy of x"

***RESPONSE*:**
Page 24-23 Sec 24.5.3.2 Clause 3) Became issue 24-002; considered and closed with no action taken.

Page 23-22 Sec 23.2.4 [lib.vector]) Became US issue 23-010; considered and closed with no action taken.

"Various trivial editorial changes") Proposed changes were accepted.

<<<<<< Public Comment #26/Clark" follows >>>>>>
February 27, 1997

My comment concerns try-blocks within exception handlers. I would like to see the C++ Standard clarify the behavior that should be expected when an exception is re-thrown in a try-block within an exception handler.

Using exception handling in my programs, I have found the handlers generally have a lot of common error processing code such as closing and/or releasing objects, resetting the object state and logging. In an effort to eliminate this redundancy, I attempted a programming structure which consolidated the common code in a single catch-all handler. This handler contains another try-block where the exception is re-thrown and caught by a more specific handler.

```
try
{
   // exception prone code here

}
catch( ... )
```

```
 7:   {
 8:      // common error code here
 9:
10:      try
11:      {
12:         throw;  // re-throw to more specific handler
13:      }
14:     catch( ExceptA& )
15:      {
16:         // handle ExceptA here
17:      }
18:      catch( ExceptB& )
19:      {
20:         // handle ExceptB here
21:      }
22:      catch( ... )
23:      {
24:         // handle unknown exception
25:      }
26:      throw;
27:   }
```

Unfortunately, the wording in the standard is not sufficient to determine whether the above example is legitimate. I am not aware of a compiler which generates an error or a warning when compiling this code. However, the question of when the temporary exception object should be deleted is apparently subject to various interpretations among compiler vendors.

Jack Reeves was kind enough to research and discuss this example in "C++ Report", Jan. '97 Vol. 9/No. 1. If the inner handlers exit without re-throwing the exception, the state of the exception object is subject to the compilers interpretation of the Standard. The throw statement at line 26 may fail because the compiler destroyed the exception when the inner handler was exited. If the throw statement on line 26 is commented out, an error may occur at line 27 if the compiler attempts to destroy the exception object a second time. (The Microsoft VC++ 4.2 compiler exhibits both of these behaviors. In contrast, the HPUX and SunSoft SPARCWorks compilers execute this example without error.)

The two relevant sections of the current Draft C++ Standard appear to be section 15.1 item 4 - "The memory for the temporary copy of the exception being thrown is allocated in an unspecified way, except as noted in 3.7.3.1. The temporary persists as long as there is a handler being executed for that exception..." and section 15.1 item 6 - "... An exception is considered finished when the corresponding catch clause exits or when unexpected() exits after being entered due to a throw.". Neither of these definitively resolve the situation out-lined above. In fact, it can be legitimately argued these statements contradict each other in this instance.

Item 4 does specifically address exiting a handler by re-throwing the exception, therefore, if the inner handlers re-throw the exception, the above example is valid. I would like to see the standard address, with similar clarity, exceptions re-thrown in a try-block within an exception handler. I believe the following questions need to be answered.

1. Should undefined behavior be expected in this instance?
2. Should the inner handler destroy the exception object even though the outer handler is still executing?

3. If the inner handler destroys the exception object,
a. Should the compiler generate an error if there is an attempt to use or re-throw the exception in the outer handler?
b. Should the outer handler also try to destroy the exception object?
c. Should the compiler recognize an invalid situation and generate an error?

Now that the problem has been explained sufficiently, I would like to turn to justifying the committees attention on the problem. First, as stated above, the rules are ambiguous in this instance and allow programs to be compiled without error or warning and then execute in an inconsistent manner.

Second, there have been newsgroup discussions about the need for adding a Java style 'finally' clause to the C++ exception handling mechanism. In fact, Robert Martin suggests this in his article in the same issue of "C++ Report" mentioned above. The arguments for a 'finally' clause are very valid except that C++ already allows a similar mechanism. Using the programming construct I have described provides functionality similar to the 'finally' clause. Common error logic is stated only once within the exception handler.

I would even argue the C++ mechanism is superior because the logic for exiting a successful routine does not have to be mixed with the logic needed to exit the routine when an error occurs. These are quite often very different and the C++ exception mechanism keeps them separated. All of the logic for normal execution is in the try block and all of the logic for error processing is in the handlers. These two execution paths do not have to merge into a 'finally' clause. The logic for normal execution can be coded without considering the error processing. Exception handlers can then be added without having to change the code for normal execution.

The problem is that most developers are not aware that the exception mechanism in C++ allows this functionality. Jack Reeves suggests using nested try-blocks as an alternative to my solution. Both approaches achieve equivalent results and seem valid to me. The standard needs to clarify the rules in this area so that valid alternatives to the 'finally' clause can be publicized and developers can choose the appropriate solution.

Steve Clark
Federated Systems Group
295 Ecarte Ct.
Lilburn, Ga. 30247
(770) 925-3820
b06swc@federated-dept-stores.com

RESPONSE: accepted -- see 15.1(except.throw) paragraph 4.

<<<<<< Public Comment #27/Jones" follows >>>>>>
February 27, 1997

While reviewing clause 2 Lexical conventions [lex] at the last J11/WG14 meeting as part of a proposal to adopt universal character names into C, I noticed something that I thought should be brought to your attention:

In subclause 2.3 Trigraph sequences [lex.trigraph], either ??? is missing from Table 1 - trigraph sequences or paragraph 4 is vacuous (since none of the trigraph replacements can occur within a trigraph) except for the example, which is completely wrong. Since ??? is not a trigraph in C and introducing it in C++ would create a gratuitous incompatibility, I suggest that paragraph 4 be deleted.

Larry Jones
SDRC
2000 Eastman Dr.
Milford, OH 45150
513-576-2070
larry.jones@sdrc.com

RESPONSE: accepted -- the example in 2.3(lex.trigraph) paragraph 4 was removed.

<<<<<< Public Comment #28/Robison/Nelson" follows >>>>>>
February 28, 1997

Arch Robison (robison@kai.com)
David Nelson (david@kai.com)
Kuck & Associates, Inc.
1906 Fox Drive
Champaign, IL 61820
(217) 356-2288

We wish to submit the following comments concerning the recently released C++ Committee Draft. Thank you for your consideration of these issues.

----

(1) In 26.4.1 [lib.accumulate], the requirements for class T are not specified. The user is left wondering what properties class T has to have to work. For example, does class T have to allow assignment? Clearly there are (recursive) implementations of accumulate that would not require assignment. But are implementors required to handle the case where T does not allow assignment. The standard should specify exactly what properties class T has to have in order to work with the accumulate template.

----

(2) The example in 14.7.3 [temp.expl.spec] paragraph 6 contradicts the last line in 14.7.3 paragraph 16. The example shows the explicit specialization syntax; the last line says that the explicit specialization syntax should not be used.

----

(3) In 27.4.2.1.1 [lib.ios::failure], method what() has a more general exception specification than the method that it is overriding. 27.4.2.1.1 says that std::ios_base::failure::what can throw any exception. But 18.6.1 [lib.exception] says that std::exception::what cannot throw any exception. This clearly contradicts 15.4

[except.spec], paragraph 3, which requires that the overriding derived-class method throw only exceptions allowed by the base-class method.

----

(4) In 26.2.6 [lib.complex.ops], paragraph 15, operator<< inserts a NUL character when writing to an ostream stream. I.e., the "as if" code shown inserts an ends, which is retained by the result of s.str() used. Did you mean s.c_str() or should the ends not be appended? Surely the intent was not to insert NUL characters into output.

----

(5) Section 21.6.1.3 defines gcount() to return the number of characters extracted by the last unformatted input function, e.g. getline(char_type *s, streamsize n). Should it also work for getline(basic_istream<charT, traits> &is, basic_string<CharT, traits, Allocator> & str) defined in section 21.3.7.9?

----

(6) In 27.6.1.3 paragraph 28, one would also expect eofbit to be set if end-of-file is encountered before n characters are stored.

----

(7) Considering section 27.8.1.4 paragraphs 1,2 and section 27.5.2.4.3 paragraphs 1-3. Should the return type of showmanyc be streamsize instead of int. Consider the case when streamsize is a 32 bit long int and int is 16 bits.

----

(8) In section 5.3.4, what happens when the allocation function does not throw an exception, but returns NULL instead. See paragraphs 16, 22 and section 18.4.1.3 and consider the following errant example:

```
      T val;
      T *p = NULL;
      new (p) T(val);
```

----

(9) In section 21.1.2 table 37, it seems that not_eof() should use eq_int_type instead of eq(). What is the behavior for integer values passed to not_eof() for which eq_int_type() returns false, but eq() returns true. For example, consider the value 0x7FFF where eof() returns 0xFFFF and a char is 8 bits.

----

(10) In section 4.10 paragraph 1, the term "nul pointer constant" is used, but in section 18.1 paragraph 4, the term "nul-pointer constant" is used.

----

(11) In 21.3.7.9 paragraph 1, start a new line before "After the last character (if any) is".

Arch Robison (robison@kai.com)
David Nelson (david@kai.com)
http://www.kai.com/C_plus_plus
KAI C++ - Cross Platform C++ Compiler

Kuck and Associates
1906 Fox Drive
Champaign, IL 61820
(217) 356-2288

***RESPONSE*:**
(1) 26.4.1 [lib.accumulate]) Became US issue 26-002; a proposed resolution was accepted.

(2) accepted -- see 14.7.3(temp.expl.spec) paragraph 5.

(3) 27.4.2.1.1 [lib.ios::failure]) Fixed specification of ios::failure::what to say that it does not throw.

(4) 26.2.6 [lib.complex.ops], paragraph 15) Became US issue 26-003; a proposed resolution was accepted.

(5) 21.6.1.3) Rejected as previously considered.

(6) 27.6.1.3 paragraph 28) Became US issue 27-008; the proposed resolution was accepted.

(7) 27.8.1.4 paragraphs 1,2 and 27.5.2.4.3 paragraphs 1-3) Became US issue 27-009; the proposed resolution was accepted.

(8) accepted -- see 5.3.4(expr.new) paragraph 13.

(9) 21.1.2 table 37, Became US issue 21-007; a proposed resolution was accepted.

(10) accepted.

(11) 21.3.7.9 paragraph 1) Became part of US issue lib-edit-001; proposed changes were accepted.

<<<<<< Public Comment #29/Shaffer" follows >>>>>>
February 28, 1997

Comments on the December 1996 C++ Draft Proposed International Standard

1) 2 Lexical conventions

The allowed uses of universal character names are not clear.

Are UCNis that name characters in the basic source character set allowed?

Are UCNis allowed in places other than: comments, identifiers, character-literals and string-literals?

2) 2.3 Trigraph sequences

While Table 1 does not include the sequence i???i, paragraph 4 implies that this is a trigraph for i?i. The following example makes this explicit.

In the ANSI/ISO C standard the i???i trigraph does NOT exist.

3) 2.11 Keywords
Table 3

The keyword "export" is not listed.

4) 3.6.1 Main function
Perhaps the signatures like the following be allowed:

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

5) 3.9.1 Fundamental types
Paragraph 1

In order to match the language with the library (which treats all characters as unsigned) plain char should be defined as unsigned.

In order to avoid breaking existing non-portable code that assumes char is signed, individual implementations could, as an extension, have an option to treat char as signed.

6) 3.10 Lvalues and rvalues
Paragraph 15

This paragraph says that an object may be accessed through an aggregate or union that has a member of the correct type. It does NOT say that you must access the object through the particular member that is of the correct type.

7) 5.2.2 Function call
Paragraph 7

When a non-POD class type is passed as an ellipsis argument the program should be ill-formed, rather than undefined. The point is to require a diagnostic from the compiler n an extension to do something useful in this case would still be possible.

8) 5.2.9 Static cast
I feel that some additional pointer conversions should be allowed by a static cast. These are conversions that are safer and more portable than the general reinterpret_cast. The conversions involved are:

1. Pointer to int converted to or from pointer to unsigned int.
2. Pointer to short converted to or from pointer to unsigned short.
3. Pointer to long converted to or from pointer to unsigned long.
4. Conversions between pointers to any two of char, signed char or unsigned char.

The last case is the one I feel most strongly about. Due to the undefined signed/unsigned status of char, it is often necessary to perform this type of cast.

Aliasing of these types is explicitly allowed by 3.10 paragraph 15, thus it seems reasonable to allow the safe new cast operator to perform the cast.

9) 14 Templates
The intended meaning of the keyword export is somewhat obscure.

Darron J Shaffer
Sr. Software Engineer
BEA Systems
(972) 738-6137
Darron.Shaffer@beasys.com

***RESPONSE*:**
1) accepted --
   1st question: No. it is an error.
   2nd question: No. only in identifiers, character literals,
   string literals and comments. See clarifications in 2.2(lex.charset)
2) accepted -- the example in 2.3(lex.trigraph) paragraph 4 was removed.
3) accepted.
4) rejected -- request for an extension.
5) rejected -- in C++ (as in C) this is implementation defined.
6) rejected -- the current wording is sufficient.
7) rejected -- An implementation can issue an error, but it's also free to issue no diagnostic and implement some kind of extension.
8) rejected -- request for an extension.
9) accepted -- the semantics of export were clarified.

<<<<<< Public Comment #30/Kuehl" follows >>>>>>
February 28, 1997

```
Name:      Dietmar Kuehl
Company:   Universitdt Konstanz
Address:   Fakultdt f|r Mathematik und Informatik
           Postfach 5560/D188
         D-78434 Konstanz
Telephone: (++49) 7531 / 88-4438
E-mail:    dietmar.kuehl@uni-konstanz.de
```

Hi,

below I have attached a bunch of comments, most of editorial nature, to the lib-locales section of the CD2. I hope that it is sufficient to submit these comments via e-mail (if not please tell me such that I can submit them by snail mail, too).

I don't know what format you would prefer. Thus, I used the following:
- first the location is specified
- the type of comment is specified
- the details follow

Regards,
  dk
--
<mailto:dietmar.kuehl@uni-konstanz.de>

01. 22.1 Locales (lib.locales)   minor bug
    The declarations of the non-member functions 'is*()' are declared to be 'const'. Although a gcc extension allows this, I don't think that it is sanctioned by the remainder of the current CD.

02. 22.1.1 Class locale (lib.locale) unnecessary restrition
    The type 'locale::category' is defined to be 'int'. I think, it

should be defined to be any bitmask type defining the corresponding values.

03. 22.1.1 Class locale (lib.locale) section 2 documentation bug
It is stated that 'use_facet' and 'has_facet' are member functions. This does not match the later definition of those two functions as non-member function templates.

04. 22.1.1 Class locale (lib.locale) section 3 example bug
In the example, the object 'cerberos' of type 'basic_ostream<...>::sentry' is constructed with a default argument but there is no default constructor for this type. Instead, it has to be constructed like

```
  typename basic_ostream<charT, traits>::sentry cerberos(s);
```

The same situation appears in other example, too.

05. 22.1.1.1.2 class locale::facet (lib.locale.facet) section 1 omission
It is missing in the definition of the static member 'id' that this member has to be either publically accessible or at least accessible to the class 'locale'. As stated, it would be legal to make the member 'private' which would not satisfy the intend (I think...).

06. 22.1.1.1.2 class locale::facet (lib.locale.facet) section 2 omission
If 'refs == 0', does this imply that the 'locale' is supposed to delete the 'facet'? If this is the case, state that the 'facet' has to be a valid argument to 'delete' (or whatevery) like it is done for the pointer managed by 'auto_ptr'.

07. 22.1.1.1.2 class locale::facet (lib.locale.facet) section 2 omission
If 'refs != 0', it is stated the the 'facet' is "deleted". This assumes that it is allocated by 'new' but I guess that the intent was to have the 'facet' be e.g. an object with static linkage: This would mean that "deleted" should be replaced by "destructed".

08. 22.1.1.2 locale ctors and dtors (lib.locale.cons) section 1 unclear
It is stated at several points that the locale has a name if some conditions are given at construction time. However, it is not clear what this name should be. Is this intentional?

09. 22.1.2 locale globals (lib.locale.globals) section 1 bug
In the "Throws" section 'this' is mentioned. This is rather strange for a global function. It should probably be replaced by 'loc'.

10. 22.1.3.1 Character classification (lib.classification) all bug
The convenience functions are all globals and thus the 'const' specification is illegal (I think).

11. 22.2.1.1 template class ctype (lib.locale.ctype) all omission
For some of the functions arguments are not named. This is no problem most of the time, just inconsistent. However, for the description of 'toupper()' I think it is an error: The [not named] argument is referenced in the description...

12. 22.2.1.1 template class ctype (lib.locale.ctype) all question
Why is explicitly 'charT*' used instead of a more general iterator? This e.g. makes it impossible to apply those functions to

'basic_string's directly since there the iterators are explicitly made "implementation defined".

13. 22.2.2.1.2 numget virtual functions (lib.facet.num.get.virtuals) section 1 documentation bug
It is stated that the operation occurs in *two* stages. This statement is immediately followed by a description of *three* stages...

14. 22.2.2.1.2 numget virtual functions (lib.facet.num.get.virtuals) section 1 error
The description of stage 2 ends with "If the character is *neither* discarded *nor* accumulated then in is advanced by ++in and processing returns to the beginning of stage 2." I think this is exactly the negation of the intended wording, i.e. this should become: "If the character is *either* discarded *or* accumulated then in is advanced by ++in and processing returns to the beginning of stage 2." I'm not 100% sure since I'm not a native English speaker...

15. 22.2.3.1.2 numpunct vritual functions (lib.facet.numpunct.virtuals) omission
in 'do_decimal_pointer()', 'do_thousends_sep()', 'do_truename()', and 'do_falsename()' objects of type 'char' are returned as 'char_type'. I think the objects returned have to be the results of 'widen()', e.g. using 'use_facet<ctype<char_type> >(locale::global())' or the same facet from a 'locale' passed as argument.

RESPONSE:

01. 22.1) Fixed.

02. 22.1.1) The type locale::category is *int* for compatibility with the C library locale-category argument, which is also *int*. This compatibility is necessary to allow C library LC_* values to be passed to the locale constructors.

03. 22.1.1) Fixed.

04. 22.1.1) Fixed.

05. 22.1.1.1.2) Fixed.

06. 22.1.1.1.2) Fixed.

07. 22.1.1.1.2) It is deleted, and the draft has been clarified.

08. 22.1.1.2) Yes, it is intentional.

09. 22.1.2) Fixed.

10. 22.1.3.1) Fixed.

11. 22.2.1.1) Fixed.

12. 22.2.1.1) Rejected as an extension.

13. 22.2.2.1.2) Fixed.

14. 22.2.2.1.2 Fixed.

15. 22.2.3.1.2) Fixed.

<<<<<< Public Comment #31/Mulhern" follows >>>>>>
February 28, 1997

My comments stem from experience attempting to implement large portions of the (draft) standard library. The observations below primarily concern the header <string> and the interconnection of that header to other parts of the library. I will forward comments to the committee regarding other standard library headers if I have time to finish bringing them into compliance with CD 14882 prior to the closing of the public comment period.

Overall, CD 14882 is an immense improvement over the previous public comment draft. The library is significantly more coherent and many of the details of library design have been worked out. I would like to thank members of the library working group for their obvious effort and care.

While implementing the header <string> I encountered the following two significant issues where I felt the draft standard was ambiguous or incorrect.

(The table numbers and paragraph numbers cited below follow those given in the printed version of CD14882; these differ from the table and paragraph numbering in the HTML version of the same document. For example, Table 37 in the paper version is Table 2 (of the strings library) in the HTML version.)

1) ISSUE: traits::eos() does not exist in Table 37, although it once was a member of the traits class; traits::eos() is nevertheless still referenced in several places in the strings library and in the I/O library.

PROBLEM DESCRIPTION: traits::eos() is either explicitly referenced or an undefined 'null character' is mentioned in the following places in the draft standard.

**** explicit references to traits::eos() ******

21.3.4 basic_string element access [lib.string.access]
In the 'Returns:" section (Paragraph 2) for the functions:
const_reference operator[](size_type pos) const;
reference operator[](size_type pos);

21.3.6 basic_string string operations [lib.string.ops]
In the 'Returns:' section(Paragraph 1) and the 'Notes:' section(Paragraph 3) for the function:
const charT* c_str() const;

27.6.1.2.3 basic_istream::operator>> [lib.istream::extractors]
In the 'Effects:' section for the functions:
template<class charT, class traits>
basic_istream<charT,traits>& operator>>(basic_istream<charT,traits>& in, charT* s);

  template<class traits>
    basic_istream<char,traits>& operator>>(basic_istream<char,traits>&
in, unsigned char* s);
  template<class traits>
    basic_istream<char,traits>& operator>>(basic_istream<char,traits>&
in, signed char* s);
In Paragraph 7: "A null byte (traits::eos()) in the next position,
which may be the first position if no characters were extracted."

27.6.1.3 Unformatted input functions   [lib.istream.unformatted]
In the 'Effects:' section for the function:
basic_istream<charT,traits>& getline(char_type* s, streamsize n,
char_type delim);
In Paragraph 20: "In any case, it then stores a null character (using
 traits::eos()) into the next successive location of the array."

27.6.2.7 Standard basic_ostream manipulators   [lib.ostream.manip]
In the 'Effects:' section for the manipulator:
     template <class charT, class traits>
          basic_ostream<charT,traits>& ends(basic_ostream<charT,traits>&
os);
Paragraph 3: "Inserts a null character into the output sequence: calls
os.put(traits::eos())."

traits::eos() is listed in the index (page 12) under 'eos,char_traits'

******* In addition to these explicit uses there are numerous references
in the library to null characters or null objects in the library where
traits::eos() would more clearly specify the intent. ******

21.3.5.7 basic_string::copy   [lib.string::copy]
In the 'Effects:' section(Paragraph 3):
       "The function does not append a null object to the string
designated by s." -- null object undefined

27.6.1.2.3 basic_istream::operator>>   [lib.istream::extractors]
In the 'Effects:' section(Paragraph 6) for the functions:
   template<class charT, class traits>
    basic_istream<charT,traits>& operator>>(basic_istream<charT,traits>&
in, charT* s);
  template<class traits>
    basic_istream<char,traits>& operator>>(basic_istream<char,traits>&
in, unsigned char* s);
  template<class traits>
    basic_istream<char,traits>& operator>>(basic_istream<char,traits>&
in, signed char* s);
    "Otherwise n is the number of elements of the largest array of
char_type that
    can store a terminating eos."   -- eos not defined.

27.6.1.3 Unformatted input functions   [lib.istream.unformatted]
In the 'Effects' section(Paragraph 8) for the function:
basic_istream<charT,traits>& get(char_type* s, streamsize n, char_type
delim);
"In any case, it then stores a null character into the next successive
 location of the array."
 --- 'null character' not defined.

21.1.1 Definitions   [lib.char.traits.defs]

In the definition of NTCTS. Here the null character is defined as charT(0). NTCTS doesn't appear in the index nor is it used anywhere in the draft standard.

21.1.2 Character traits requirements [lib.char.traits.require]
In Table 37--Traits requirements, for the expression 'X::length(p)', the Postcondition is: "yields: the smallest i such that X::eq(p[i],charT(0)) is true." which would imply that charT(0) was the 'null character' for all charT.

21.3.3 basic_string capacity [lib.string.capacity]
For the function : void resize(size_type n); Paragraph 7: 'Effects resize(n,charT())." - would seem to imply that the null character is charT().

Finally, the 'null character' is used throughout the strings library indirectly in the form of references to traits::length().

FORCES:

We want to have a unique end-of-string character for all possible charT, not just char and wchar_t.

Such an end-of string character is a throw back to null terminated char*'s and we would prefer not to have to define it at all.

There should be only one definition of the null character, not the current three: charT(), charT(0) and traits::eos().

RESOLUTION:

To my mind traits::eos() provides the appropriate level of generality that should exist in the standard library. I believe charT() and charT(0) references should be replaced by references to traits:eos(). traits::eos() will have to be added to the requirements for traits in Table 37. If traits::eos() was included in the traits requirements users of the library could redefine traits::eos() to be other than the expected 0-character for basic_string< char >, thus enabling strings with embedded 'nulls' for special applications. In Table 37, X::eos() should be a user defined end_of_string character. Only the char_traits specializations for char and wchar_t should define eos() to be (char)0 or (wchar_t)0, as appropriate. Users can override the default with their own traits classes.

Whatever definition the committee settles upon for the 'null character', the draft standard should use that definition consistently and replace the conflicting definitions.

2) ISSUE: basic_string< charT, traits, Allocator >::npos is used in expressions where max_size() should be used.

EXAMPLE:

21.3.1 basic_string constructors [lib.string.cons]
For the constructor:
basic_string(const charT* s, size_type n, const Allocator& a = Allocator());
Paragraphs 6 & 7 are:
"Requires:
s shall not be a null pointer and n < npos.

```
  Throws:
    out_of_range if n == npos."
```

Should be:

```
"Requires:
    s shall not be a null pointer and n <= max_size().
  Throws:
    out_of_range if n  > max_size()."
```

Comments: If you look in '21.3.3 basic_string capacity [lib.string.capacity]' at the function void resize(size_type, charT) you'll see that this same thing done correctly (once) in the current draft. For this resize function the implementation is described in part as: "Requires: n <= max_size() Throws: length_error if n > max_size()." Moreover, there doesn't seem to be any particular reason to prohibit 'n==npos' specifically other than the fact that it won't succeed. The real limit on allocation is max_size(). One supposes that this is six of one and half a dozen of another, that is, if the condition for throwing an out_of_range exception involves npos as is currently stated in the draft then the user will inevitably get a bad_alloc exception for all n > max_size() if not an out_of_range exception. I believe that the expression should involve max_size() as shown above to be an effective error indication.

OTHER INSTANCES: The same reasoning applies to the following instances of conditions on a throw statement, in addition to the instance cited above.

****

21.3.1 basic_string constructors [lib.string.cons]
For the constructor:
basic_string(size_type n, charT c, const Allocator& a = Allocator());
Paragraphs 12 & 13 contain:

```
  "Requires:
    n < npos
  Throws:
    length_error if n == npos."
```

Should be:

```
"Requires:
    n < = max_size()
  Throws:
    length_error if n  > max_size()."
```

****

21.3.5.2 basic_string::append [lib.string::append]
For the function: basic_string<charT,traits,Allocator>& append(const basic_string<charT,traits>& str, size_type pos, size_type n);
In Paragraph 4: "The function then throws length_error if size() >= npos - rlen."
Should be "The function then throws length_error if size() > max_size() - rlen."

****

21.3.5.4 basic_string::insert [lib.string::insert]
For the function: basic_string<charT,traits,Allocator>& insert(size_type pos1, const basic_string<charT,traits,Allocator>& str,

size_type pos2, size_type n);
In Paragraph 4: "Then throws length_error if size() >= npos - rlen."
Should be: "Then throws length_error if size() > max_size() - rlen."

****

21.3.5.6 basic_string::replace   [lib.string::replace]
For the function: basic_string<charT,traits,Allocator>&
replace(size_type pos1, size_type n1, const
basic_string<charT,traits,Allocator>& str, size_type pos2, size_type n2
);
In paragraph 4: "Throws length_error if size() - xlen >= npos - rlen."
Should be: "Throws length_error if size() - xlen > max_size() - rlen."

****

With the above issues out of the way, I would indulge in a modest enhancement to class basic_string and some of the container classes. The current definition of the class basic_string contains the member function 'basic_string<charT,traits,Allocator>& erase(size_type pos = 0, size_type n = npos)' which reduces the size() of a string. It also contains the member function 'void resize(size_type n, charT c)' which for 'n < size()' has the same effect as if 'erase(n, npos)' were called. Thus we have redundant ways to reduce the size() of a string. Meanwhile, for basic_string, and also for the sequences vector, list and deque, we have the following dilemma: suppose, in the basic_string case, that we want to read in variable length lines from a text file into a basic_string< char >. In order to read in the lines of the text file without incurring reallocation overhead I might want to reserve() a large amount of memory up front for each string that I read in. After reading in a string using, say, the global function getline() I might want to 'shrink' the allocation down to the actual size() of the string read. Right now the draft standard gives me no way to do this with class basic_string. But, here comes the enhancement, if we added language to the definition of resize() such that for 'n <= size()' implementations were permitted to reduce the allocation for the string such that capacity() might be reduced to as little as n. This functionality might be even more useful with the sequences vector, list and deque. Offhand, I can think of many occasions when I wanted this shrinking capability for class vector when acquiring data from a database. This enhancement does not effect any of the already stated effects of resize() for any of the classes mentioned. That's all the enhancements I have to offer.

Finally, a list of what I believe are cut and paste errors or typos:

1) 21.3.1 basic_string constructors   [lib.string.cons]
For the constructor:

```
basic_string( const basic_string<charT,traits,Allocator>& str,
                      size_type pos = 0, size_type n = npos,
                      const Allocator& a = Allocator() );
```

In Table 39, remove the line "get_allocator()   str.get_allocator()"
This is a holdover from a previous version of this constructor which didn't take its own Allocation& argument but instead used str.get_allocator().

2) 21.3.5.6 basic_string::replace   [lib.string::replace]
Missing template parameters on the return value basic_string's.
The function "basic_string& replace(iterator i1, iterator i2, const basic_string& str);"

should be: "basic_string<charT,traits,Allocator>&
            replace(iterator i1, iterator i2, const basic_string& str);"
The function "basic_string& replace(iterator i1, iterator i2, const
charT* s, size_type n);"
should be: "basic_string<charT,traits,Allocator>&
                   replace(iterator i1, iterator i2, const charT* s,
size_type n);"
The function "basic_string& replace(iterator i1, iterator i2, const
charT* s);"
should be: "basic_string<charT,traits,Allocator>&
                   replace(iterator i1, iterator i2, const charT* s);"
The function "basic_string& replace(iterator i1, iterator i2, size_type
n, charT c);"
should be: "basic_string<charT,traits,Allocator>&
               replace(iterator i1, iterator i2, size_type n, charT c);"
The function "template<class InputIterator>
                       basic_string& replace(iterator i1, iterator i2,
InputIterator j1, InputIterator j2); "
should be: "template<class InputIterator>
                   basic_string<charT,traits,Allocator>&
                   replace(iterator i1, iterator i2, InputIterator j1,
InputIterator j2); "

3) 21.3.6.8 basic_string::compare [lib.string::compare]
The function "int compare(const basic_string<charT,traits,Allocator>&
str)" should be const as declared in '21.3 Template class basic_string
[lib.basic.string]' at Paragraph 4.
The signature should be
"int compare(const basic_string<charT,traits,Allocator>& str) const"

4) 20.4.4.3 uninitialized_fill_n [lib.uninitialized.fill.n]
 template <class ForwardIterator, class Size, class T>
 void uninitialized_fill_n(ForwardIterator first, Size n, const T& x);
The 'Effects:' section is simply incorrect; currently it is:
 " Effects:
  while (n--)
      new (static_cast<void*>(&*result++))
          typename
iterator_traits<ForwardIterator>::value_type(*first++);"
This is erroneous. It must be:
" Effects:
  while (n--)
      new (static_cast<void*>(&*first++))
           typename iterator_traits<ForwardIterator>::value_type(x);"

5) 27.4.2.3 ios_base locale functions [lib.ios.base.locales]
For the function 'locale imbue(const locale loc);'
There are extraneous characters in the 'Returns:' section at the line
"output operations.La Postcondition: loc == getloc()."
Remove the extraneous 'La'.

I hope these comments have been useful. I look forward to the
completion of the C++ Standard.

John Mulhern
Euler Solutions
945 Bayless Avenue
Saint Paul, Minnesota
55114
(612)525-8915

email: jfm@euler.com

***RESPONSE*:**
1) Became US issue 21-002; a proposed resolution was accepted.

2) Became issue 21-008; considered and closed with no action taken.

"Finally, a list of what I believe are cut and paste errors or typos") Portions became part of US issue lib-edit-001; proposed changes were accepted.

<<<<<< Public Comment #32/Aldridge" follows >>>>>>
March 4, 1997

Name: John Aldridge
Company: Graphic Data Systems

Address: Wellington House
         East Road
         Cambridge
         CB1 1BH
         ENGLAND

Phone: +44 1223 371925
E-mail: jpsa@uk.gdscorp.com

The committee draft seems deficient in the statements it makes about the validity of iterators and references into STL containers. The only statements I can find are:

   23.2.1.3 on insert and erase in deques
   23.2.2.3 on insert and erase in lists
   23.2.4.2 on reallocation on vectors
   23.2.4.3 on insert & erase in vectors

I can find no statement on whether other methods on containers result in the invalidation of iterators or references to containers.

In particular, for associative containers, I'd expected (hoped) to find a statement such as:

> insert does not affect the validity of iterators and references
> to the container, and erase invalidates only the iterators and
> references to the erased elements

which is taken from the Stepanov & Lee STL document, "The Standard Template Library", dated October 31, 1995.

Together with one (applying to all containers) such as:

> Unless otherwise stated (either explicitly or by defining a
> function in terms of the application of other functions),
> invoking a member function of a container or passing a container
> as argument to a container library function will not cause

➢ references or iterators to that container to become invalid.

RESPONSE: Became US issue 23-011; a proposed resolution was accepted.

<<<<<< Public Comment #33/Miller" follows >>>>>>
March 4, 1997

Name: Randy D. Miller
Company Name: Self
Address: 20684 SW Teton Ave, Tualatin, Oregon, 97062-8814, USA
Voice Phone: 503-692-2863
Email: tango@teleport.com

Comment:

Section [dcl.stc]/2, last sentence, references section [stmt.expr] because "expression statements" are mentioned. Proposal: it is more important to reference section [stmt.ambig] because it is ambiguity resolution that is specifically being discussed.

RESPONSE: Accepted.

<<<<<< Public Comment #34/Miller" follows >>>>>>
March 4, 1997

Name: Randy D. Miller
Company Name: Self
Address: 20684 SW Teton Ave, Tualatin, Oregon, 97062-8814, USA
Voice Phone: 503-692-2863
Email: tango@teleport.com

Comment:

The document is ambiguous whether functions and member functions are "objects." Section [intro.defs] does not define "object" at all. Section [intro.object] defines "object" as "a region of storage" which is "created by a definition ... or by the implementation (12.2) when needed." That would necessarily include functions. Nothing else in [intro.object] excludes functions or member functions from being objects." However, in [basic.types]/1, types are said to describe objects, references, or functions, implying that the set of function types is disjoint from the set of object types. Elsewhere in the document, functions seem to be tacitly excluded when discussing "objects."

Suggestion: add language to make it clear if function types are "objects" or not.

RESPONSE: rejected -- although 1.7(intro.object) paragraph 1 was clarified.

<<<<<< Public Comment #35/Miller" follows >>>>>>
March 4, 1997

Name: Randy D. Miller
Company Name: Self
Address: 20684 SW Teton Ave, Tualatin, Oregon, 97062-8814, USA
Voice Phone: 503-692-2863
Email: tango@teleport.com

Comment:

Apparent typographical omission: to correct, insert the word "or" immediately before the words "for the copy of an object thrown..." in 3.7.3.1(4) [basic.stc.dynamic.allocation]/4.

RESPONSE: Editorial.

<<<<<< Public Comment #36/Miller" follows >>>>>>
March 4, 1997

Name: Randy D. Miller
Company Name: Self
Address: 20684 SW Teton Ave, Tualatin, Oregon, 97062-8814, USA
Voice Phone: 503-692-2863
Email: tango@teleport.com

Comment:

Propose that [expr.call]/9 be changed from
"Recursive calls are permitted."
to:
"Recursive calls are permitted, except to the function named 'main' ([basic.start.main]/3])."

This will resolve a contradiction between [basic.start.main]/3 which prohibits main() from being called from within a program, and [expr.call]/9 which permits recursive calls.

RESPONSE: Accepted.

<<<<<< Public Comment #37/Holle" follows >>>>>>
March 5, 1997

Re: type_info::name()'s specification in ANSI C++ Draft

type_info::name()'s usefulness is _severely_ limited by the statement in the standard that its return value is implementation defined. If it were defined, then general, full-featured, cross-platform persistence and storage mechanism could be easily implemented based on it. With implementation defined behavior, however, it is not clear how this can be achieved.

Allen Holub authored an MSJ article in June 96 showing how easy a persistence mechanism would be if type_info::name() had standardized behavior. He and I both agree, however, that his mechanism does not work across 2 compilers according to the current draft.

I realize that the standardization of these return values is tedious and annoying to the standard's committee, BUT it is imperitive to the utility of this portion of the language! Allen is now using Java rather than C++ for just such reasons. I would be too if native Java compilers were available.

I therefore _strongly_ urge that type_info::name()'s return values are standardized.

--
Jess Holle
Senior Software Engineer
Parametric Technology Corporation
(617) 398-5015
jessh@ptc.com

RESPONSE: Rejected -- request for an extension. There is some interest in generating a Technical Report on the subject to be published after the Standard is published.

<<<<<< Public Comment #38/Lilley" follows >>>>>>
March 5, 1997

I have some comments regarding the exception-specification clause for functions and function-pointers. It seems that excluding the exception-specification from the declarator for pointer-to-pointer-to-function opens up some loopholes for defeating the exception-specification checking that occurs when function-pointers are assigned.

In particular, the throw(int) here is disallowed:

```
int (**pf)() throw(int);
```

The reason, as far as I can tell, comes from 15.4.12:

"An exception-specification is not considered part of a function's type"

Since the exception-specification is not part of the type, it is pointless to include the exception-specification in contexts that are only using the type of the function. But that is inconsisent with the language that says the exception-specification of pointers-to-functions *is* meaningful, and must be checked during assignment from one function-pointer to another. In my opinion, it was a mistake to exclude the exception-specification from the type of the function. Making the exception-specification part of the function type would have made things more consistent.

For example, this is a problem because it defeats the exception-specification checking:

```
void f_throw() throw(int);
void f_nothrow();
```

```
    void (*fp_nothrow)();
    void (*fp_throw)() throw (int);
    void (**fpp)();

    fp_nothrow = f_throw;     // (1) OK, less restrictive
    fp_throw = f_nothrow;     // (2) error, more restrictive
    fpp = &fp_nothrow;        // (3) OK??  double-indirection has
                              // no exception-specification.
    fp_throw = *fpp;          // OK?? Didn't this defeat (2)?
```

Of course, I do not have a compiler to verify the above assertions, but they seem to be true, given the current language.

respectfully submitted,

John Lilley

jlilley@empathy.com

Nerds for Hire, Inc.
4270 Evans Dr.
Boulder, CO 80303

RESPONSE: rejected -- It was considered at length and finally rejected by the committee.

<<<<<< Public Comment #39/Choolinin" follows >>>>>>
March 6, 1997

I offer to include in STL template the following template of the function:

```
template< class _to, class _from >
_to safe_cast( _from value )
{
      assert( value == _from( _to( value )  )  );
      return _to( value );
}
```

This function allow to convert numbers or text strings or anything else from format to format without danger to lose information.

Eg:

```
      short index();
      short next_index = safe_cast< short, int >( index() + 1 );
```

S. Y. George G. Choolinin, BITSoftware, Inc. Programmer, Moscow, Russia
Jura@bitsoft.ru

RESPONSE: Rejected as extension.

<<<<<< Public Comment #40/Buck" follows >>>>>>
March 6, 1997

There is a mismatch between the specification of the container adaptors "stack" and "queue" and the specification of the container classes. The result that, while the standard specifies in [lib.queue]

"Any sequence supporting operations front(), back(), push_back() and pop_front() can be used to instantiate queue."

and in [lib.stack]

"Any sequence supporting operations back(), push_back() and pop_back() can be used to instantiate stack."

and in [lib.priority.queue]

"Any sequence with random access iterator and supporting operations front(), push_back() and pop_back() can be used to instantiate priority_queue."

these statements cannot be satisfied unless either the signatures of certain functions are changed, or additional requirements are imposed on sequences. That is, the current spec is self-contradictory.

Specifically, the following functions

```
stack<T,Container>::top()
queue<T,Container>::front()
queue<T,Container>::back()
priority_queue<T,Container,Compare>::top()
```

have return values of type Container::value_type& . But they are defined as c.front() or c.back() on the underlying Container c, and these functions are defined as being of type Container::reference_type, which may or may not be equal to Container::value_type&. Requiring Container::value_type& forbids containers to use "smart reference" objects, or allocators that use such objects.

There is a simple solution; I will illustrate it for stack, and the corresponding change will work for queue and priority_queue.

```
namespace std {
    template <class T, class Container = deque<T> >
    class stack {
    public:
      typedef typename Container::value_type      value_type;
      typedef typename Container::reference       reference; // CHANGE
      typedef typename Container::const_reference const_reference; //
CHANGE
      typedef typename Container::size_type       size_type;
      typedef typename Container                  container_type;
    protected:
      Container c;
    public:
      explicit stack(const Container& = Container());

      bool      empty() const             { return c.empty(); }
      size_type size()  const             { return c.size(); }
      reference       top()               { return c.back(); } // CHANGE
```

```
      const_reference top() const         { return c.back(); } // CHANGE
      void push(const value_type& x)      { c.push_back(x); }
      void pop()                          { c.pop_back(); }
    };
    ...
};
```

Note that top() is now correct for all legal containers that satisfy the conditions. Note also that no code will break, because for the cases that work with the SGI and HP STL implementations (those where the reference type is a true reference) the type of top() does not change.

The analogous change should be made to queue<..>::front(), queue<...>::back(), and priority_queue<...>::top().

Thank you.

Joseph Buck   jbuck@synopsys.com
Synopsys, Inc.   Phone: +1 415 694 1729
700 E. Middlefield Rd.   Fax: +1 415 694 1626
Mountain View, California 94043

RESPONSE: The committee agreed it was a good idea, but due to an administrative accident this comment got lost and was not rediscovered until it was too late to deal with it. Closed with no action for now.

<<<<<< Public Comment #41/Clarke" follows >>>>>>
March 6, 1997

David L. Clarke.
25 Walbridge Hill Road, P.O. Box 328, Tolland, CT 06084-0328

Mitakuye oyasin,
David L. Clarke
davec@imagine.com

I work (full time) for Pratt & Whitney Aircraft, Mail Stop 161-05, 400 Main St., East Hartford, CT 06108; (part time) for Rensselaer at Hartford, 275 Windsor St., Hartford, CT 06120-2991; and I am also a self employed author writing a book on systems programming.

My phone numbers are home: (860) 872-7653; work: (860) 565-9395

My e-mail addresses are: davec@imagine.com; clarkedl@pweh.com; davec@hgc.edu.

I recently attempted to move a working program developed in my capacity as an educator to the platform we will be using at work (Pratt & Whitney). The program originally ran under Borland C++ and Microsoft Visual C++, both of which used the STL and strings libraries of the draft proposal. The target platform was a DEC alpha running Digital UNIX. To my surprise, the port did not compile.

I traced the problem to a change that has been made to the strings library definition. The original code used the string::remove() method, which followed the draft proposal of 28 April 1995. The DEC compiler expects string::erase(), which follows the December 1996 draft proposal. Apparently "remove()" was removed and replaced.

I understand that erase() is more comparable to the corresponding method for STL classes such as list, but removing "remove()" will cause a lot of existing code to no longer compile. I feel that the remove() form should be retained to avoid this problem. I have used remove() in both my teaching materials and in examples I have used in my book. I would not like to have to re-do all of this work.

Perhaps remove() and erase() can both be kept and used as synonyms.

Thank you and
mitakuye oyasin, (Lakota for "we are all related")

David Clarke

RESPONSE: Previously considered and rejected.

<<<<<< Public Comment #42/Choolinin" follows >>>>>>
March 7, 1997

An discussions, wich took plase at BITSof, result the folowing improvement of template safe_cast:

```
template< class _to >
safe_cast {
public:
      template< class _from >( _from arg ) : body( arg ) {
            assert( _from( body ) == arg ); }
      operator _to() const { return body; }
private:
      _to body;
};
```

usage of this class is the same to the privious offer, but type _from always is correct, and there is not possibility to lose data by miss choose of type _from:

```
short f();
/* use class -- all correct */
short next_f = safe_cast< short >( f() + 1 );

/* use function -- posible misstake: */
short next_f = safe_cast< short, char >( f() + 1 );
```

S. Y. George G. Choolinin, BITSoftware, Inc. Programmer, Moscow, Russia
Jura@bitsoft.ru

RESPONSE: Rejected as an extension.

<<<<<< Public Comment #43/Abrahams" follows >>>>>>
March 7, 1997

As someone who is not (yet) a member of the C++ committee, I figure I should make a public comment about my pet issue, just to be sure it gets addressed. I know this makes extra work for the committee, and I apologize in advance. Since I am on the libraries mail reflector and plan to attend at least the Nashua meeting, I hope that whoever is issuing responses will be able to minimize their efforts -- I'll already be somewhat informed.

The issue is that the current standard makes it unreasonably hard to write exception-safe programs using the standard library templates. In particular, the current standard only details some of the conditions under which the library is liable to produce undefined behavior, e.g. crash (see 17.3.3.6)! Many of the conditions are produced by obvious and useful combinations of library components, such as vector<string>.

In order to write exception-safe programs, library clients need a "contract" provided by the library which guarantees predictable behavior if clients fulfill their part of the bargain. This contract should provide the following (at least):

1. Certainty that the library does not leak resources. In particular, every contained object constructed by a container should be destroyed by the time that container is destroyed. Also, functions such as uninitialized_fill must destroy the objects they have constructed if any construction fails.

2. Certainty that the library maintains any invariants guaranteed by the implementation of its contained objects' public interface. For example, if use of a contained object's public interface maintains the object's destructibility, the library will do the same. Or, if a contained object's assignment operator implements "commit-or-rollback" semantics, the objects in a container will always be complete copies of objects constructed outside the container.

3. A way to get "commit-or-rollback" semantics from containers. This is critical to exception recovery. If the contents of containers are unpredictable after an exception is thrown, it becomes impossible to maintain long-lived containers to support a running program. For example, a program may need to maintain a vector of multiprocessing tasks. If an exception is thrown while inserting a new task, it may be important that the vector's state hasn't changed. If a program can't count on the integrity of its long-lived containers, the best it can do in response to an exception is unwind the stack and exit. In that case, why bother with exceptions at all?

Thanks,
David

David Abrahams * Mark of the Unicorn, Inc. * abrahams@motu.com

---

RESPONSE: Became issue 23-005 "Library containers lack exception policy". Library changes were accepted to resolve the issue.

<<<<<< Public Comment #44/DeRocco" follows >>>>>>
March 11, 1997

I'd like to see void be usable as a valid type for a template parameter, so that generated functions could include void among their possible return types.

I'd also like to see "void" be usable as the type of an external identifier, meaning that the identifier refers to something at some address whose type is unknown. The only thing you'd be able to do with the identifier would be to take its address, which would naturally be of type void*.

--

Paul D. DeRocco
DeRocco Engineering
87 Duff St.
Watertown, MA 02172
617-923-8987
mailto:pderocco@ix.netcom.com

RESPONSE:
1) no action -- already allowed.
2) rejected -- request for an extension.

<<<<<< Public Comment #45/DeRocco" follows >>>>>>
March 11, 1997

The standard defines offsetof, which nicely complements sizeof. I'd like to see lengthof added to the standard, which would return the number of elements in an array:

```
#define lengthof(x) (sizeof(x) / sizeof(*(x)))
```

It is obviously undefined for some arguments, and produces meaningless results for others, but the same is true for offsetof.

--

Paul D. DeRocco
DeRocco Engineering
87 Duff St.
Watertown, MA 02172
617-923-8987
mailto:pderocco@ix.netcom.com

RESPONSE: Rejected as an extension.

<<<<<< Public Comment #46/Brown" follows >>>>>>
March 12, 1997

My apologies if this is to late or addressed to the wrong person.

I noticed the following error in the Dec 96 draft C++ standard.
In section 20.1.1 Equality comparison [lib.equalitycomparable]
the last line of the following table reads

```
--If a == b and b == a, then a == c
```

I assume that it should read

```
--If a == b and b == c, then a == c
                     ^
```

Cheers,

Steve Brown

Stephen Brown   Phone (61)(2) 94126018
CSIRO Telecommunications and Industrial Physics   Fax   (61)(2) 94133293
126 Greville Street, Chatswood NSW 2067, Australia
sbrown@ul.rp.csiro.au
http://www.ul.rp.csiro.au

RESPONSE: Accepted as editorial change.

<<<<<< Public Comment #47/Parker" follows >>>>>>
March 12, 1997

Name: Brian Parker
Business Name: Voxel Graphics Research
Email: bparker@gil.com.au
Postal Address:
14 Jules Av,
Rochedale South,
Brisbane QLD 4123,
Australia

Phone: +61 7 32992807

Comments by section follow.

Page 7-16 Sec 7.3.3 [namespace.udecl]
I think that this section doesn't properly specify the interaction
between using declarations and templates within namespaces. In
particular, for the following example,

```
namespace test{
      template<class T>
      class tt{
      };
}
```

It is not clear that the following is allowed (though I certainly hope it is).

```
using test::tt;

tt<int> a;
tt<double> b;
```

(For example: Microsoft VC4.2 doesn't accept the using test::tt but it is accepted by Borland 5.01.)

And is, "using test::tt<int>" allowed such that only that particular specialisation is accessible, and if so does it require a template instantiation?

I think that at least one of the namespace examples should include a nested template to make the intent clearer.

Page 14-36 Sec 14.8.2 [temp.deduct]
I think that the template argument(s) within nested template instantiations should be deducible.

For example, T should be deducible in

```
template<class T>
void f( A< B<T>, C<T> >);
```

where A, B & C are previously defined template classes of 2, 1 & 1 parameters respectively.

The draft standard clause 9 states that T can be deduced for class-template-name<T>, but the term "class-template-name" is not defined anywhere. The parameter to the above function is essentially a template type of a single parameter T and so it should not be any more difficult for an implementation to deduce T for this than for a simple non-nested template type, so defining "class-template-name" to mean only non-nested template types would be an unnecessary limitation. It should be defined to mean any parameterized type. (Note: this is not a purely theoretical issue; I personally have code that would benefit from this.)

This may already be the intent of the draft standard; in any case, "class-template-name" needs to be defined.

Page 14-36 Sec 14.8.2 [temp.deduct]
Given several function arguments where the template argument can be deduced using one of the arguments but not the others, then shouldn't that deduction be used for the template function call? Clause 10 allows this in the specific case of a nested type definition, but clause 2 would seem to disallow it in general ("If type deduction cannot be done for any parameter/argument pair ... deduction fails."). Is there any need for this restriction?

Various trivial editorial changes:

Page 27-4 Sec 27.2 [lib.iostream.forward] clause 9.
Semicolon left off end of class char_traits definition.

Page 20-11 Sec 20.3.6.1 [lib.binder.1st].
In class binderfirst, change "Operation::first_argument_type value" to "typename Operation::first_argument_type value".
and the same for Page 20-12 [lib.binder.2nd].

Page 12-4 Sec 12.2 [class.temporary] clause 5. Example at top of page.
From "friend const C& operator+" to "friend const C operator+" or "friend C operator+"

Page 20-1 Sec 20.1.1 [lib.equality.comparable] Table 28.
From "If a == b and b == a, then a == c." to "If a == b and b == c, then a == c."

***RESPONSE*:**
1) rejected -- the grammar already prohibits a using declaration that names a template specialization.
2) no action -- the requested support is already provided.
3) rejected -- this approach was considered too error-prone.
"Various trivial editorial changes") Referred to project editor.

<<<<<< Public Comment #48/Galichsky" follows >>>>>>
March 12, 1997

Name: Konstantin V.Galichsky
Company name: PHYSICON, Ltd.
Address: BOX 59, Dolgoprudny-1 Moscow region, Russia, 141700
Telephone number: +7 (095) 408-77-72
Email: kg@scph.mipt.ru

Comment to [dcl.fct.spec] and [class.virtual]:

The syntax for virtual function definition and overriding is the same:

```
    class Base {
        // Introduce new entry in vtable.
        virtual void f ();
    };

    class Derived : public Base {
        // Replace the entry in vtable, but syntax is the same.
        virtual void f ();
    };
```

Assume that a programmer makes some misprint in the name or in the parameter list of Derived::f. The compiler will not detect this error, instead, it will introduce new entry in the vtable! Furthermore, it is not easy for a program's reader (a human) to resolve the declaration of a new virtual function from the overriding of some "old" existing one. This produces new source of not-easy-to-detect errors and makes text not easy to read.

My proposal:

C++ must support additional syntax for the virtual function overriding, for instance:

```
    class Derived : public Base {
        override void f ();
    };
```

In this case, the compiler must check the existance of the definition of virtual f() in direct or indirect base classes. In addition, this makes the code more readable. The old syntax, of course, must be preserved for compatibility.

Drawback of the proposal:

The new keyword may break existing code. Another solution is to use some existing keywords, for instance:

```
    class Base {
        // The 'new' keyword guarantees that it is firstly defined.
        new virtual void f ();
    };

    class Derived : public Base {
        // 'Continue' guarantees that it is overriding of Base::f ().
        continue virtual void f ();
    };
```

RESPONSE: rejected -- request for an extension.

<<<<<< Public Comment #49/DeRocco" follows >>>>>>
March 15, 1997

1) C++, like C, is rather limited in its initializer list syntax. In particular, you can leave things uninitialized (or zeroed, if static), or you can type out your initializers one at a time. Why not allow ... to be used at the end of an array initializer list, meaning that the last value should be repeated to the end of the array?

```
      bool flags[1000] = { true, ... };
```

This doesn't break any existing legal programs, and the alternative is either lots of typing, or explicit initialization code. It is also clean and well-defined. Of course, the compiler would be free to translate the above into some initialization code to fill the array--as it would have to if the array were local to a function.

2) It would also be extremely useful if one could specify that initializer sublists be repeated a certain number of times. A syntax that wouldn't conflict with any existing legal programs would be to follow a brace-enclosed list with an asterisk and a constant expression:

```
      int foo[200] = { { 0, 1 } * 50, { 2, 3 } * 50 };
```

There may be some gotchas in here--I haven't had time to think this one through, but the March 18 deadline is right around the corner, so I thought I'd mention it anyway.

3) Once an initializer can be implicitly repeated in either of the above two ways, the need to base an initialization value on the position within the array becomes apparent. Since the name of the array has no useful meaning within the initializer list, I suggest that within the list the array name be interpreted as a constant of type size_t whose value is the current array index being initialized. For instance:

```
      long squares[1000] = { squares * squares, ... };

      long table[][2] = // table of squares and cubes
              { { table * table, table * table * table } * 1000 };
```

You might kick this one around.

--

Paul D. DeRocco
DeRocco Engineering
87 Duff St.
Watertown, MA 02172
617-923-8987
mailto:pderocco@ix.netcom.com

RESPONSE: rejected -- request for an extension.

<<<<<< Public Comment #50/Galichsky" follows >>>>>>
March 15, 1997

Name: Konstantin V.Galichsky
Company: PHYSICON, Ltd.
Address: BOX 59, Dolgoprudny-1 Moscow Region, Russia, 141700
Phone: +7 (095) 408-77-72
Email: kg@scph.mipt.ru

In the very beginning of "temp" the export keyword is introduced. This keyword is absent in the list of keywords ("lex.key").

RESPONSE: Accepted.

<<<<<< Public Comment #51/DeRocco" follows >>>>>>
March 15, 1997

Unlike C, C++ frequently involves the use of objects that need to be named for syntactic reasons, but whose names are never subsequently used. It would be useful if there was a reserved word that the compiler replaces with a guaranteed-unique machine-generated name whenever it occurs. Possible choices would be "unnamed", "_" or "__". I sort of prefer "_" because it's short.

For instance, here's an object that calls a function when it's constructed and another function when it's destroyed:

```
typedef void (*vfunc)();

class initializer {
    const vfunc term;
public:
    initializer(vfunc i, vfunc t): term(t) { if (i) (*i)(); }
    ~initializer() { if (term) (*term)(); }
};
```

This can be used to force the execution of a function before and/or after main():

```
void init_func() { ... }
void term_func() { ... }

initializer _(init_func, term_func);      // doesn't need name
```

Another common example is an object representing ownership of a resource:

```
class lock {
    mutex& mut;
public:
    lock(mutex& m): mut(m) { m.capture(); }
    ~lock() { mut.release(); }
};

mutex the_mutex();

void func() {
    lock _(the_mutex);   // doesn't need name
    ...
}
```

The freedom not to make up a specific name for the object becomes particularly important when there are loads of such invocations, or when the invocations are generated by a template or preprocessor macro.

--

Paul D. DeRocco
DeRocco Engineering
87 Duff St.
Watertown, MA 02172
617-923-8987
mailto:pderocco@ix.netcom.com

RESPONSE: rejected -- request for an extension.

<<<<<< Public Comment #52/Jorgensen" follows >>>>>>
March 18, 1997

I have a comment on the scope of names declared in for statements.

In the working paper: section 6.5.3 "The for statement", subsection 3:

"If the for-init-statement is a declaration, the scope of the name(s) declared extends to the end of the for-statement."

This is not the same as "The C++ prog. lang." (second edition), reference manual r.6.5.3, last sentence:
"If the for-init-statement is a declaration, the scope of the names declared extends to the end of the block enclosing the for-statement."

The cause of this difference is the difference between the "equivalent statements":

```
  (Working paper, section 6.5.3, subsection 1):
        {
            for-init-statement
            while ( condition ) {
                statement
                expression ;
            }
        }
  (The C++ prog. lang, second edition, reference manual r.6.5.3):
        for-init-statement
        while ( expression-1) {
            statement
            expression-2 ;
        }
```

Problem: This causes formerly well-formed programs to become ill-formed.

For example:

```
  for(int i=0; e[i]!=0; i++)
    ;
  e[i] = a;     //ill-formed according the the working paper
```

For another example,

```
  for(int i=0; i<10; i++)
    a[i] = 0;
  for(i=0; i<20; i++) //ill-formed according the the working paper
    b[i] = 0;
```

From a fresh viewpoint I would probably like the definition in the working paper (limiting the scope of names to the for statement) the most. However, from the viewpoint of the person forced to modify several sources I would like the new standard to stick with the old definition (extending the scope of names beyond the for statement).

Yours sincerely,
Ivan Skytte Jorgensen
eur!con!isj@aask.dk

RESPONSE: rejected. The change in semantics was introduced in 1992. No objections have been raised since then which were not previously considered.

<<<<<< Public Comment #53/Houlder" follows >>>>>>
March 18, 1997

Tom Houlder
Dagaliveien 13
0387 Oslo

Norway
47-22 14 57 71
thoulder@pemail.net

and has been formulated with the help of
James Youngman, JYoungman@vggas.com
****

SUGGESTION

I suggest that the class `complex' is changed so that the member functions `T complex::real()' and `T complex::imag()' are suppressed and the data `T real' and `T imag' are introduced as public data.

Alternatively, `T complex::real()' and `T complex::imag()' could be retained for the convenience of existing code, but be made to return a reference `T&' instead. New public variables, with for instance the names `T re' and `T im', ought nevertheless to be added to the class.

*****

MOTIVATION

The suggested change makes it possible to explicitly modify the data without instantiating a new `complex' object.

The change also reflects the mathematical nature of a complex number, which can be thought of as two standard numbers on the real line (just like a point in the x-y plane) subject to special mappings. The implication is that the `complex' class should be regarded as a container class (a pair) of two numbers.

The suggested change will make `complex' consistent with the rest of the standard library as can be seen by the following:

One-dimensional variables are represented by `double's (or `float's or `int's, etc). These are trivially not subject to protected access. N-dimensional variables are represented by `valarray's (or `vector's or `deque's) of `double's. These are not subject to protected access either as there are operators like

```
reference operator[](size_type n)
```

returning a non-constant reference so that the number can be changed directly.

So why should a two-dimensional variable hide its components? It is difficult to find other reasons than that it "looks good" to a C++ programmer and that, theoretically, some obscure optimisation can be done for the other functions interacting with the class. On the other hand, there are weighty reasons for not hiding the data. It is tempting to mention numerous examples from physics or mathematics where it is necessary to manipulate the real and the imaginary parts directly without any overhead. However, that is not needed. Just imagine how it would be if changing the i'th element of a `vector' had to be done by performing an operation with another `vector' or by creating a new one. Alternatively, imagine all the problems we would have if it were impossible to alter a standard variable otherwise than by calling operator functions on it or by creating a new one. One thing is sure, the application in question would be severely slowed down.

It is important to understand that there is no difference between these examples and the present `complex' class for programmers who work with time critical applications requiring complex numbers. The `complex' class should be a container class holding two numbers, not a class encapsulating the fictive entity "A complex number". From a mathematical point of view, such an entity does not exist other than in daily speech, and it should absolutely not be included in the only object oriented language which is useful for time critical scientific applications. The class `pair' is a perfect example of how the data in a `complex' class should be represented.

*****

CONCLUSION

The `complex' class coming with the standard library is of very limited use in its present implementation since the individual data can not be manipulated directly. The simple suggested change will dramatically increase the class's usability at apparently no loss.

*****

Sincerely yours,

Tom Houlder

*****

APPENDIX

Suggested changes illustrated by changing the section 26.2.2 of CD2-ASCII.

[Primary Suggestion:

```
  26.2.2  Template class complex                           [lib.complex]
  namespace std {
    template<class T>
    class complex {
    public:
      typedef T value_type;

      T real;
      T imag;

      complex(const T& re = T(), const T& im = T());
      complex(const complex&);
      template<class X> complex(const complex<X>&);

      complex<T>& operator= (const T&);
      complex<T>& operator+=(const T&);
      complex<T>& operator-=(const T&);
      complex<T>& operator*=(const T&);
      complex<T>& operator/=(const T&);

      complex& operator=(const complex&);
      template<class X> complex<T>& operator= (const complex<X>&);
      template<class X> complex<T>& operator+=(const complex<X>&);
      template<class X> complex<T>& operator-=(const complex<X>&);
      template<class X> complex<T>& operator*=(const complex<X>&);
```

```
    template<class X> complex<T>& operator/=(const complex<X>&);
  };

template<class T> complex<T> operator+(const complex<T>&, const T&);
template<class T> complex<T> operator+(const T&, const complex<T>&);
template<class T> complex<T> operator-(const complex<T>&, const T&);
template<class T> complex<T> operator-(const T&, const complex<T>&);
template<class T> complex<T> operator*(const complex<T>&, const T&);
template<class T> complex<T> operator*(const T&, const complex<T>&);
template<class T> complex<T> operator/(const complex<T>&, const T&);
template<class T> complex<T> operator/(const T&, const complex<T>&);
template<class T> complex<T> operator==(const complex<T>&, const T&);
template<class T> complex<T> operator==(const T&, const complex<T>&);
template<class T> complex<T> operator!=(const complex<T>&, const T&);
template<class T> complex<T> operator!=(const T&, const complex<T>&);
```

1 The class complex describes an object that stores the Cartesian components, T real and T imag, of a complex number.]

[Secondary Suggestion:

26.2.2 Template class complex [lib.complex]

```
namespace std {
  template<class T>
  class complex {
  public:
    typedef T value_type;

    T re;
    T im;

    T& real();
    T& imag();
    const T& real() const;
    const T& imag() const;

    complex(const T& x = T(), const T& y = T());
    complex(const complex&);
    template<class X> complex(const complex<X>&);

    complex<T>& operator= (const T&);
    complex<T>& operator+=(const T&);
    complex<T>& operator-=(const T&);
    complex<T>& operator*=(const T&);
    complex<T>& operator/=(const T&);

    complex& operator=(const complex&);
    template<class X> complex<T>& operator= (const complex<X>&);
    template<class X> complex<T>& operator+=(const complex<X>&);
    template<class X> complex<T>& operator-=(const complex<X>&);
    template<class X> complex<T>& operator*=(const complex<X>&);
    template<class X> complex<T>& operator/=(const complex<X>&);
  };

template<class T> complex<T> operator+(const complex<T>&, const T&);
template<class T> complex<T> operator+(const T&, const complex<T>&);
template<class T> complex<T> operator-(const complex<T>&, const T&);
template<class T> complex<T> operator-(const T&, const complex<T>&);
template<class T> complex<T> operator*(const complex<T>&, const T&);
```

```
  template<class T> complex<T> operator*(const T&, const complex<T>&);
  template<class T> complex<T> operator/(const complex<T>&, const T&);
  template<class T> complex<T> operator/(const T&, const complex<T>&);
  template<class T> complex<T> operator==(const complex<T>&, const T&);
  template<class T> complex<T> operator==(const T&, const complex<T>&);
  template<class T> complex<T> operator!=(const complex<T>&, const T&);
  template<class T> complex<T> operator!=(const T&, const complex<T>&);
```

1 The class complex describes an object that stores the Cartesian components, T re and T im, of a complex number.]

RESPONSE: Closed without further action; previously considered.

<<<<<< Public Comment #54/Neyman" follows >>>>>>
March 24, 1997

It seems that there is some inconsistency in definition of trigraph sequences in December 1996 Draft (2.3, lex.trigraph). The table of trigraph sequences contains 9 elements, and it is explicitly stated that no other trigraph sequences exist. "???" is not present in the table of trigraph sequences.

However,a subsequent example states that the sequence "???=" becomes "?=". This seems to indicate that "???" is a trigraph sequence that should be replaces with "?". If this is so, then this sequence should probably be added to the table of trigraph sequences.

- Vladimir Neyman
Dow Jones Telerate
vlad@tts.telerate.com
201-938-5790

RESPONSE: accepted -- the example in 2.3(lex.trigraph) paragraph 4 was removed.