Doc No: X3J16/97-0010
WG21/N1048
Date: January 28th, 1997
Project: Programming Language
C++
Ref Doc:
Reply to: Josee Lajoie
          (josee@vnet.ibm.com)

```
+=======================+
| Core WG List of Issues |
+=======================+
```

The first half of this document contains the substantive and editorial core issues.

Because the core list of issues was not published in the post-Hawaii mailing, the issues that were closed at the Hawaii meeting are listed at the end of this document.

```
+--------+
| Syntax |
+--------+
9.2 [class.mem]:
  692: ";opt" after member "function-definition" should be omitted


+-----------------+
| C Compatibility |
+-----------------+
1.1 [intro.scope]:
  604: Should the C++ standard talk about features in C++ prior to 1985?
Annex C:
  680: Annex C subclause C.1 is out of date
  743: Some anachronisms are missing from annex C
Annex E:
  770: The title of Annex E needs to be made shorter


+--------------------+
| Lexical Conventions |
+--------------------+
2.3 [lex.trigraph]:
  744: Is the description of trigraph processing wrong?


+-------+
| Core1 |
+-------+

Conformance model
-----------------
1.7 [intro.compliance]:
  602: Are ill-formed programs with non-required diagnostics really
       necessary?
  619: Is the definition of "resource limits" needed?

Name Look Up
------------
3.3.6 [basic.scope.class]:
  664: When does the reevaluation rule for class scope name lookup
require a
       diagnostic?
3.4.2 [basic.lookup.koenig]:
    686: Where is a function name looked up if an argument type is
introduced
         with a typedef or a using-declaration?
  3.4.3 [basic.lookup.qual]:
    665: In X::~Y is Y looked up in the context of the current expression?
  3.4.5 [basic.lookup.classref]:
    688: Rules for name lookup after :: . -> need to be clarified for
         conversion-function-id, template argument names and destructor
names

  Linkage / ODR
  -------------
  3.2 [basic.def.odr]:
    745: Does &inline_function yield the same result in all the translation
         units?
  7.5 [dcl.link]:
    729: Must extern "C" functions declared in a namespace and a global
extern
         "C" function have different signatures and return types?
    749: Can a declaration specify both a storage class and a linkage
         specification?
    750: To which declarator in a member function declaration does the
         extern "C" specifier apply?
  9.5 [class.union]:
    505: Must anonymous unions declared in unnamed namespaces also be
static?

  Object/Memory Model
  -------------------
  3.6.2 [basic.start.init]:
    746: What is the order of initialization of a class static data member?
    747: The term "static initialization" needs to be defined
  5.3.4 [expr.new]:
    669: semantics for new and delete expressions should be separated from
the
         requirements for operator new and delete
    690: Clarify the lookup of operator new in a new expression
  5.7 [expr.add]:
    720: Can you do &*p if p does not point to a valid object?
  5.9 [expr.rel]:
    721: Comparisons of pointer to class members need fine tuning
  5.19 [expr.const]:
    722: The definition of address constant expression needs fine tuning
  7.3.3 [namespace.udecl]:
    672: using-declarations and base class assignment operators
  8.5 [dcl.init]:
    751: Should { } be allowed around an initializer that is a string?
  10.1 [class.mi]:
    624: class with direct and indirect class of the same type: how can the
         base class members be referred to?
  12.4 [class.dtor]:
    753: Is 'new char[size]' aligned properly to hold an object of any type
T?
  12.5 [class.free]:
    754: for new T, allocation functions in base classes of T are not
         considered
  12.8 [class.copy]:
    687: The WP prohobits the copy assignment of virtual base classes to
behave
         like the copy constructor
    755: Assignment of POD class objects: is the class copied as a block?


  +-------+
  | Core2 |
  +-------+

  Sequence Points/Execution Model
  -------------------------------
  1.8 [intro.execution]:
    603: Do the WP constraints prevent multi-threading implementations?
    694: List of full-expressions needed

  Access
  ------
  11.5 [class.protected]:
    752: When accessing a base class member, the qualification is not
ignored

  Types / Classes / Unions
  ------------------------
  3.9 [basic.life]:
    621: The terms "same type" need to be defined

  Default Arguments
  -----------------
  8.3.6 [dcl.fct.default]:
    689: What if two using-declarations refer to the same function but the
         declarations introduce different default-arguments?
    730: When are default arguments for member functions of template
classes
         semantically checked?


  Types Conversions / Function Overload Resolution
  ------------------------------------------------
  4.1 [conv.lval]:
    711: Is an lvalue-to-rvalue conversion on an incomplete type allowed
         within a sizeof operand?
  4.8 [conv.double]:
    712: Is an lvalue-to-rvalue conversion on an incomplete type allowed
         within a sizeof operand?
  5.2.2 [expr.call]:
    713: What argument type can be passed to va_arg?
    714: Is the term "default argument promotions" needed?
  5.4 [expr.cast]:
    718: Conversion to and from pointers to incomplete class types using
old
         style casts - is this really implementation-defined?
  7.2 [dcl.enum]
    683: What is the underlying type of an enumeration type if the value of
an
         enumerator uses the value of a previous enumerator?
  13.3.3.1 [over.best.ics]:
    733: Implicit conversion sequences and scalar types
  13.6 [over.built]:
    682: operator ?: and operands of enumeration types
    734: ambiguity in "bool & ? void *& : classType&" where classType has
an
         operator void*&
    756: most uses of built-in "?" with class operands are ambiguous

  Expressions
  -----------
  5 [expr]:
    748: Should we say that operator precedence is derived from the syntax?
  5.6 [expr.mul]:
    719: Is unsigned arithmetic modulo 2~N for multiplication as well?
  +--------+
  | Core 3 |
  +--------+

  Templates
  ---------
  14 [temp]:
    757: Can a template member function be overloaded?
  14.3 [temp.arg]:
    758: Can an array name be a template argument?
    759: Initializing a template reference parameter with an argument of a
         derived class type needs to be described
    760: Is a template argument that is a private nested type accessible in
         the template instantiation context?
  14.5.1.1 [temp.mem.func]:
    761: Can the member function of a class template be virtual?
  14.5.5.1 [temp.arg]:
    762: How can function template be overloaded?
  14.5.5.2 [temp.func.order]:
    763: Partial Specialization: the transformation also affects the
function
         return type
  14.6 [temp.res]:
    736: How can/must typename be used?
    764: undeclared name in template definition should be an error
    765: The syntax does not allow the keyword 'template' in
         'expr.template C<parm>'
    766: How do template parameter names interfere with names in nested
         namespace definitions?
  14.6.4 [temp.dep.res]:
    737: How can dependant names be used in member declarations that appear
         outside of the class template definition?
  14.6.4.1 [temp.point]:
    767: Where should the point of instantiation of class templates be
         discussed?
  14.8.2 [temp.deduct]:
    677: Should the text on argument deduction be moved to a subclause
         discussing both function templates and class template partial
         specializations?
    768: typename keyword missing in some examples

  Exception Handling
  ------------------
  15.2 [except.dtor]:
    769: Are the base class dtors called if the derived dtor throws an
         exception?

  ==========================================================================
====
   Chapter 1 - Introduction
  ------------------------
  Work Group:     Core
  Issue Number:   604
  Title:          Should the C++ standard talk about features in C++ prior
to
                  1985?
  Section:        1.1 [intro.scope]
  Status:         editorial
  Description:
          UK issue 229:
          "Delete the last sentence of 1.1 and Annex C.1.2. This is the
first
```

standard for C++, what happened prior to 1985 is not relevant to this document."

Resolution:

At the Hawaii meeting, the C compatibility WG decided:
"Delete references to C.1. Annex C.1 needs to be removed or rewritten."

Requestor: UK issue 229
Owner: (C Compatibility)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 602
Title: Are ill-formed programs with non-required diagnostics really necessary?
Section: 1.3 [intro.compliance]
Status: active
Description:

UK issue 9:
"We believe that current technology now allows many of the non-required diagnostics to be diagnosed without excessive overhead. For example, the use of & on an object of incomplete type, when the complete type has a user-defined operator&(). We would like to see diagnostics for such cases."

Question: Do deprecated features render a program ill-formed but no diagnostic is required?

See also UK issue 93.

Resolution:
Requestor: UK issue 9
Owner: Josee Lajoie (Conformance Model)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 619
Title: Is the definition of "resource limits" needed?
Section: 1.3 [intro.compliance]
Status: editorial
Description:

1.3 para 2 says:
"Every conforming C++ implementation shall, within its resource limits, accept and correctly execute well-formed C++ programs..."
The term resource limits is not defined anywhere.
Is this definition really needed?

Resolution:
Requestor: ANSI Public comment 7.12
Owner: Josee Lajoie (Conformance Model)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 603
Title: Do the WP constraints prevent multi-threading implementations?

```
  Section:        1.8 [intro.execution]
  Status:         active
  Description:
          UK issue 11:
          "No constraints should be put into the WP that preclude an
           implementation using multi-threading, where available and
           appropriate."

          Bill Gibbons notes:
          For example, do the requirements on order of destruction between
          sequence points preclude C++ implementations on multi-threading
          architectures?
  Resolution:
  Requestor:      UK issue 11
  Owner:          Steve Adamczyk (sequence points)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   694
  Title:          List of full-expressions needed
  Section:        1.8 [intro.execution]
  Status:         editorial
  Description:
          1.8p14: "certain contexts in C++ cause the evaluation of a
          full-expression that results from a syntactic construct other
          than expression"

          Is it enumerated anywhere exactly what these contexts are?
          Do the contexts themselves at least identify themselves as
          surrogate full-expressions?

          I didn't read the cited example (8.3.6) as thoroughly as I
          might, but I didn't see anything there that explicitly said
          "this is treated like a full-expression." Probably you could
          make the case based on combining several passages together, but
          if the other ones are like this, it would take some real
          detective work to figure it out.  If someone knows what contexts
          were intended here, even if something might be omitted, it would
          be an improvement to make it explicit, either here or in the
          various contexts.
  Resolution:
  Requestor:      Mike Miller
  Owner:          Steve Adamczyk (Sequence Points)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  =========================================================================
====
   Chapter 2 - Lexical Conventions
  ---------------------------------
  Work Group:     Core
  Issue Number:   744
  Title:          Is the description of trigraph processing wrong?
  Section:        2.3[lex.trigraph]
  Status:         active
  Description:
          2.3 para 4 says:
          "Trigraph replacement is done left to right, so that when two
           sequences which could represent trigraphs overlap, only the
           first sequence is replaced. [Example: The sequence "???="
           becomes "?=", not "?#". The sequence "?????????" becomes
           "???", not "?". -- end example]"

          [Clark Nelson, edit-778:]

          > A new paragraph was added after the September draft,
          > specifically [lex.trigraph]/4. The paragraph seems to be
          > trying to clarify some aspects of trigraph processing.
          >
          > Unfortunately, the entire paragraph seems to be based on a
          > false premise; to wit, that ??? is a trigraph which is
          > replaced by a single ?.  However, ??? is not listed as a
          > trigraph sequence in the trigraph table, and according to
          > paragraph 3, there are no other trigraphs. If ??? were
          > a trigraph for ?, then paragraph 4 would be meaningful and,
          > arguably, necessary clarification. However, if (as I believe)
          > ??? is not a trigraph of any sort, then the new paragraph 4
          > is actually meaningless and/or just plain wrong, and should be
          > deleted.
          >
          > As a possibly related issue, in the C standard, the statements
          > of paragraph 3 are normative. Should the note-brackets around
          > that paragraph be removed from the working paper? If they were,
          > the confusion about ??? might have been a little less likely.
  Resolution:
  Requestor:      Clark Nelson
  Owner:          Tom Plum (Lexical Conventions)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  ========================================================================
===
   Chapter 3 - Basic Concepts
  ---------------------------
  Work Group:     Core
  Issue Number:   745
  Title:          Does &inline_function yield the same result in all the
                  translation units?
  Section:        3.2[basic.def.odr]
  Status:         editorial
  Description:
          3.2 para 4 says:
          "An inline functions shall be declared in every translation unit
in
           which it is used."
          It is not clear from this statement whether taking the address
          of an inline function in different translation units must yield
          the same result.
  Resolution:
  Requestor:
  Owner:          Josee Lajoie (ODR)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   664
  Title:          When does the reevaluation rule for class scope name
lookup
                  require a diagnostic?
  Section:        3.3.6 [basic.scope.class]
  Status:         editorial
  Description:
          3.3.6 para 1 says:
```

1) The potential scope of a name declared in a class consists not only of the declarative region following the name's declarator, but also of all function bodies, default arguments, and constructor ctor-initializers in that class (including such things in nested classes).

2) The name N used in a class S shall refer to the same declaration when re-evaluated in its context and in the completed scope of S.

3) If reordering member declarations in a class yields an alternate valid program under (1) and (2), the program's behavior is ill-formed, no diagnostic is required.

According to the wording above, a diagnostic is required to be issued for the following program. Should it?

```
typedef int I; //1

class D {
    typedef I I; //2
};
```

This is ill-formed according to rule 2) but not according to rule 3) (i.e. this not a reordering problem). Rule 3) is the rule for which "no diagnostic is required."

Should Rule 2) also say: "no diagnostic is required."? Otherwise, this will require that an implementation processes class member declarations twice in order to determine if names used by the declaration change meaning.

Resolution:

Rule 2) was modified to say:
"No diagnostic is required for a violation of the rule."
The example above should be added to the WP.

Requestor: Steve Adamczyk
Owner: Josee Lajoie (Name Lookup)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 686
Title: Where is a function name looked up if an argument type is introduced with a typedef or a using-declaration?
Section: 3.4.2 [basic.lookup.koenig]
Status: editorial
Description:

basic.lookup.koenig says:

When an unqualified name is used as the postfix-expression in a function call (_expr.call_), other namespaces not considered during the usual unqualified look up (_basic.lookup.unqual_) may be searched; this search depends on the types of the arguments.

For each argument type T in the function call, there may be a set of zero or more associated namespaces to be considered; such namespaces are determined in the following way:

[...]
- If T is a class type, its associated namespaces are the namespaces in which the class and its direct and indirect base classes are defined.

This text is not very clear as to what happens if the type was introduced with a typedef or a using-declaration:

```
namespace N1 {
        struct T { };
        void f(T);
        void g(T);
};

namespace N2 {
        using N1::T;
        typedef N1::T U;

        void f(T);
        void g(U);
};

void foo() {
        N2::T t;
        N2::U u;

        f(t);           // which f?
        g(u);           // which g?
}
```

Resolution:

The following was added to 3.4.2 paragraph 2:
"Typedef names used to specify the types do not contribute to this set."

I still think some text should be added to say what happens if the type was introduced with a using declaration.

Requestor: Andrew Koenig
Owner: Josee Lajoie (Name Lookup)
Emails: core-7041
Papers:

. . . .

Work Group: Core
Issue Number: 665
Title: In X::~Y is Y looked up in the context of the current expression?
Section: 3.4.3 [basic.lookup.qual]
Status: active
Description:

In an expression like

```
p->X::~X();
```

where is the "X" that follows the "~" looked up?

3.4.5 [basic.lookup.classref] says that in an unqualified name, the name after the ~ is looked up in the current context and in the class of p. But it doesn't say anything special about the qualified case.

This implies that it is looked up in the scope of X only. If this is true, it seems to me that is a problem because it doesn't work when X is a typedef, as in:

```
struct A {
        ~A();
};

typedef A AB;

int main()
{
        AB *p;
        p->AB::~AB();
}
```

This suggests that the name after ~ should always be looked up in the current context, even for the qualified name case. Presumably, for the qualified name case it would also be looked up in the class of the qualifier.

Resolution:
Requestor: John Spicer
Owner: Josee Lajoie (Name Look Up)
Emails:
Papers

. . . .

Work Group: Core
Issue Number: 688
Title: Rules for name lookup after :: . -> need to be clarified for conversion-function-id, template argument names and destructor names
Section: 3.4.5 [basic.lookup.classref]
Status: active
Description:

How is
o a destructor name
o an id-expression of a conversion-function-id
o a template-id
o the name of a template-argument
looked up when used following a nested-name-specifier or a class member access operator . or -> .

Bill Gibbons provided the following table, which I [Josee] filled up:

expression	name to look up	look in surrounding context	must be visible there ?	look in what class	must be visible there
A::b	b	no	---	A	yes
A::~T	T	no	---	A	yes
A::Z::~T	Z	no	---	A	

					yes
A::Z::~T	T	no	---	A::Z	yes
A::operator T	T	no	---	A	yes
A::operator Z::T	Z	no	---	A	yes
A::operator Z::T	T	no	---	A::Z	yes
A::C<D>	C	no	---	A	yes
A::C<D>	D	yes	yes	no	---
A::X::b	b	no	---	A::X	yes
A::X::~T	T	no	---	A::X	yes
A::X::Z::~T	Z	no	---	A::X	yes
A::X::Z::~T	T	no	---	A::X::Z	yes
A::X::operator T	T	no	---	A::X	yes
A::X::operator Z::T	Z	no	---	A::X	yes
A::X::operator Z::T	T	no	---	A::X::Z	yes
A::X::C<D>	C	no	---	A::X	yes
A::X::C<D>	D	yes	yes	no	---
a.b	b	no	---	A	yes
a.~T	T	yes	yes	A	yes
s.~T	T	yes	yes	---	---
a.operator T	T	yes	yes	A	yes
a.operator Z::T	Z	yes	yes	A	yes
a.operator Z::T	T	no	---	Z	yes
a.C<D>	C	no	---	A	yes
a.C<D>	D	yes	yes	no	---
a.X::b	X	yes	no	A	no
a.X::b	b	no	---	X	yes
a.X::~T	T	no	---	A::X	yes
s.X::~T	T	yes	yes	---	---
a.X::operator T	T	no	---	A::X	yes
a.X::operator Z::T	Z	no	---	A::X	yes

```
      a.X::operator Z::T     T          no           ---       A::X::Z
yes
      a.X::C<D>              C          no           ---       A::X
yes
      a.X::C<D>              D          yes          yes       ---       --
-

      where a is an object of class type A
      where s is an object of scalar type
```

We have to clarify the WP to ensure that the above resolutions are clear.

Bill also raises the following issues:

* The current rules for lookup of "T" in "a.operator T" break template because "T" must be visible in the class, which is impractical if "T" is a template type parameter. I propose changing the rule so the lookup is in the surrounding context only, as with template-id arguments.

* The current rules for lookup of "X" in "a.X::b" break templates because when "T" is a template type argument, the instantiation will fail if some base class of "A" (which might itself be a template type argument) happens to have a typedef or class member "T". This might be fixed as a special case in template name lookup, but I propose the simpler fix of changing the rule so the lookup is in the surrounding context only.

Resolution:
Requestor: Bill Gibbons
Owner: Josee Lajoie (Name Lookup)
Emails: core-6969
Papers

. . . .

Work Group: Core
Issue Number: 746
Title: What is the order of initialization of a class static data member?
Section: 3.6.2[basic.start.init]
Status: editorial
Description:

> On comp.std.c++, jlilley@empathy.com (John Lilley) writes:
> The order of construction is determined by the placement of
> the *definitions* of the static members, not the
> declarations within the containing class. Within a single
> translation unit (source file), the static members are
> constructed in the order of definition (DWP s3.6.2.1).

Perhaps it is an oversight, rather than a deliberate omission, but section 3.6.2/1 in the Nov 96 working paper refers to "objects of namespace scope with static storage duration"; it does not mention objects of _class scope_ with static storage duration (i.e. static members).

As far as I can tell, the current wording of the draft leaves the order of initialization of static members unspecified.

Resolution:
Requestor: Fergus Henderson
Owner: Josee Lajoie (Object Model)

```
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   747
  Title:          The term "static initialization" needs to be defined
  Section:        3.6.2[basic.start.init]
  Status:         editorial
  Description:
          para 2 says:
          "An implementation is permitted to perform the initialization
           of an object of namespace scope with static storage duration
           as a static initialization..."

          The term 'static initialization' needs to be defined.
  Resolution:
  Requestor:
  Owner:          Josee Lajoie (Object Model)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   621
  Title:          The terms "same type" need to be defined
  Section:        3.9 [basic.types]
  Status:         editorial
  Description:
          The WP needs to define what it means for two objects/expressions
          to have the same type. The phrase is used a lot throughout the
WP.
  Requestor:
  Owner:          Steve Adamczyk (Types)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  ======================================================================
====
   Chapter 4 - Standard Conversions
  ----------------------------------
  Work Group:     Core
  Issue Number:   711
  Title:          Is an lvalue-to-rvalue conversion on an incomplete type
                  allowed within a sizeof operand?
  Section:        4.1 [conv.lval]
  Status:         editorial
  Description:
          4.1 Paragraph 1 says:
            "An lvalue ...  can be converted to an rvalue.  If T is an
             incomplete type, a program that necessitates this conversion
             is ill-formed."
          Paragraph 2 says:
            "When an lvalue-to-rvalue conversion occurs within the
             operand of sizeof (5.3.3) the value contained in the
             referenced object is not accessed, since that operator does
             not evaluate its operand."

          It isn't entirely clear from this whether it is OK to have an
          lvalue-to-rvalue conversion on an incomplete type within a
          sizeof operand.  And if we can, what does it mean.

          In general, the WP is somewhat vague on which restrictions are
```

```
        relaxed in a sizeof operand.
  Resolution:
  Requestor:      Bill Gibbons
  Owner:          Steve Adamczyk (Type Conversions)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   712
  Title:          Should the result value of a floating-point conversion be
                  implementation-defined?
  Section:        4.8 [conv.double]
  Status:         active
  Description:
          4.8 says for floating-point conversions:
            If the [floating-point] source value is between two adjacent
            [floating-point] destination values, the result of the
            conversion is an unspecified choice of either of those values.

          yet 2.13.3 says for floating-point literals:

            the result is either the nearest representable value, or the
            larger or smaller representable value immediately adjacent to
            the nearest representatble value, chosen in an
            implementation-defined manner.

          Why not say "implementation-defined" for conversions too?

          This also applies to the integral to floating conversions
          described in 4.9 [conv.fpint].
  Resolution:
  Requestor:      Bill Gibbons
  Owner:          Steve Adamczyk (Type Conversions)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  ==========================================================================
====
   Chapter 5 - Expressions
  ------------------------
  Work Group:     Core
  Issue Number:   748
  Title:          Should we say that operator precedence is derived from
the
                  syntax?
  Section:        5[expr]
  Status:         editorial
  Description:
          para 4:
          "Except where noted, the order of evaluation of operands of
           individual operators and subexpressions of individual
           expressions, and the order in which side effects take place, is
           unspecified."

          "Except where noted"
          Should we say that operator precedence is derived from the
          syntax? The C syntax says this in a footnote. (Footnote 35).
  Resolution:
  Requestor:
  Owner:          Steve Adamczyk (Expressions)
  Emails:
  Papers:
```

```
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   713
  Title:          What argument type can be passed to va_arg?
  Section:        5.2.2 [expr.call]
  Status:         editorial
  Description:
          5.2.2/7 says:
          "The lvalue-to-rvalue (4.1), array-to-pointer (4.2), and
           function-to-pointer (4.3) standard conversions are performed
           on the argument expression.  After these conversions, if the
           argument does not have arithmetic, enumeration, pointer,
           pointer to member, or class type, the program is ill-formed."

          What else can it be?  Is this really meaningful?
          Wouldn't be more explicit to say which argument is _disallowed_.
  Resolution:
  Requestor:      Bill Gibbons
  Owner:          Steve Adamczyk (Type Conversions)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   714
  Title:          Is the term "default argument promotions" needed?
  Section:        5.2.2 [expr.call]
  Status:         editorial
  Description:
          5.2.2/7 says:
          "These promotions are referred to as the default argument
           promotions."

          This may be the ISO C name, but it is very confusing in C++.
          It makes one ask, why are only default arguments promoted?
          Can we use a different name?

          Steve Adamczyk:
          > It was added so it could be referenced in the 18.7
          > description of va_start, instead of repeating the words, but
          > that didn't happen.
  Resolution:
  Requestor:      Bill Gibbons
  Owner:          Steve Adamczyk (Type Conversions)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   669
  Title:          semantics for new and delete expressions should be
                  separated from the requirements for operator new and
                  delete
  Section:        5.3.4 [expr.new], 5.3.5 [expr.delete]
  Status:         editorial
  Description:
          Erwin Unruh wrote a paper (96-0011/N0829) that suggested that the
          semantics for the new expression and the delete expression be
          reworked so that they would only describe which operator new (or
          operator delete) they call.  The restrictions on the behavior of
the
          allocation and deallocation functions called should be moved to
the
```

library section.

Subclause 5.3.4[expr.new] and 5.3.5[expr.delete] still has some troublesome passages.

5.3.4 New

o Paragraph 8, last sentence says:
  "The pointer returned by the new-expression is non-null and
   distinct from the pointer to any other object."

The part of this sentence that says "and distinct from the pointer to any other object" should be deleted. This is really a requirement on the library operator new. Maybe a note should be added to say: "If the library allocation function is called, the pointer returned is distinct from the pointer to any other object."

o Paragraph 13, first sentence says:
  "The allocation function shall either return null or a pointer
   to a block of storage in which space for the object shall have
   been reserved."

This sentence should be moved to the note that follows. Again, this is a requirement that applies to the semantics of the library operator new and should not be in the normative text for 5.3.4.

Also paragraph 13 should be moved after paragraph 10, which discusses allocation functions.

o Paragraph 16 says:
  "The allocation function can indicate failure by throwing a
   bad_alloc exception (_except_, _lib.bad.alloc_). In this case
   no initialization is done."

This should be changed to:
"If the allocation function exits by throwing an exception, no
 initialization is done."

o Paragraph 21 says:
  "The way the object was allocated determines how it is freed:
   if it is allocated by ::new, then it is freed by ::delete,
   and if it is an array, it is freed by delete[] or ::delete[]
   as appropriate."

This should be deleted. Name lookup in 5.3.4 and 5.3.5 indicate which operator new and delete is called.

5.3.5 Delete

o Paragraph 2, the last few sentences say:
  "In the first alternative (delete object), the value of the
   operand of delete shall be a pointer to a non-array object
   created by a new-expression, or a pointer to a sub-object
   (_intro.object_) representing a base class of such an object
   (_class.derived_). If not, the behavior is undefined. In the
   second alternative (delete array), the value of the operand of
   delete shall be a pointer to the first element of an array
   created by a new-expression. If not, the behavior is undefined.
   [Note: this means that the syntax of the delete-expression must

match the type of the object allocated by new, not the syntax of
the new-expression.]"

The requirements that the object (or array) must be created by a new-expression should be removed. If a user operator delete is called, and this operator does nothing, then all is fine.

o Paragraph 7 says:
"To free the storage pointed to, the delete-expression will call a
deallocation function (_basic.stc.dynamic.deallocation_)."

"To free the storage pointed to," should be removed. Again, whether the storage is freed depends on which operator delete is called. A user operator delete may not free the storage.

Resolution:
Requestor: Erwin Unruh
Owner: Josee Lajoie (Memory Model)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 690
Title: Clarify the lookup of operator new in a new expression
Section: 5.3.4 [expr.new]
Status: editorial
Description:

5.3.4 should describe the lookup of operator new in a new expression.

Here is an interesting example:

```
struct C {
        operator void* new(size_t);
        operator void* new[](size_t);
};

... new C[N1][N2]; // which operator new is called?
```

Resolution:
Requestor:
Owner: Josee Lajoie (Memory Model)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 718
Title: Conversion to and from pointers to incomplete class types using old style casts - is this really unspecified?
Section: 5.4 [expr.cast]
Status: active
Description:

p6 describes conversions to and from pointer to incomplete class type and it says:
"whether the static_cast or reinterpret_cast interpretation is used is unspecified."

Since static_cast does not allow incomplete types, does this mean that it's unspecified whether old-style casts allow conversion between pointers to incomplete types?

Mike believes this should not be left unspecified but should be clearly specified by the standard as being ill-formed; i.e. the static_cast interpretation is chosen.

Resolution:
Requestor: Mike Miller
Owner: Steve Adamczyk (Type Conversions)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 719
Title: Is unsigned arithmetic modulo 2~N for multiplication as well?
Section: 5.6 [expr.mul]
Status: editorial
Description:

5.6/3, Binary * operator

According to 3.9.1/3, unsigned arithmetic is always modulo 2^N. For addition and subtraction this is easy to remember, but for multiplication the rule should probably be repeated here since it is less obvious.

Resolution:
Requestor: Bill Gibbons
Owner: Steve Adamczyk
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 720
Title: Can you do &*p if p does not point to a valid object?
Section: 5.7 [expr.add]
Status: active
Description:

5.7p5:
"If the result is used as an operand of the unary * operator, the behavior is undefined unless both the pointer operand and the result point to elements of the same array object, or the pointer operand points one past the last element of an array object and the result points to an element of the same array object, or the pointer operand points to the element of an array and the result points one past the last element of the same array."

Mike Miller proposes to remove this wording.
He says:

> All the cases described as giving undefined behavior if the result is used as the operand of unary * are already undefined behavior according the preceding sentence, regardless of how the result is used.

Bill Gibbons:

> Yes, but there still needs to be some editorial work here. There should be a description of how a "one past the end" pointer can be used.
>
> For example:
>
> ```
>    void f() {
>        int x[3];
>        int *p = x + 3;
>        int &rx = *p;    // defined behavior?
> ```

```
>        int y = rx[-1];
>    }
>
> There have been some changes in the last year which allow the
> limited use of an lvalue for an incomplete object type.  There
> are at least three related situations for valid pointers which
> do not refer to objects of the pointed-to type:
>
> * "(*p)", where "p" points just past the end of an array
>
> * "(*p)", where "p" points to zero-length array as in "p =
>           new int[n]" when "n" is zero.  This is a variation
>           of the above, since the start of the array and the
>           "just past the end" point are the same.
>
> * "(*p)", where p is zero.
>
> Consider each of these in the context of "q = &*p".
>
> I think the first two should have the expected defined
> behavior.  The last case is questionable, but there may be
> good reason to allow it.
>
> The current WP already supports 99% of this proposal.
>
> The following example is now well-formed, even if "q" is
> initialized before "x":
>
>   // translation unit #1
>   extern int p;
>   int *q = &*p;
>
>   // translation unit #2
>   int f();
>   int x = f();
>   int *p = &x;
>
> So we have the concept of an lvalue which refers to raw
> memory, suitably aligned, where the lvalue can be manipulated
> as long as the uninitialized value is never used.
>
> (A similar example could be constructed using a direct call
> to operator new and a deferred call to placement new
> "new (p) int" where the raw memory does not have a type
> explicitly associated with it.)
>
> Since a pointer to the end of an array is suitable aligned,
> the memory and object models almost support the proposal
> today.
>
> The only difference is whether it is required that a block of
> raw memory to which an lvalue refers (but does not access),
> and the address of which is a valid pointer, must actually
> exist.
>
> (Plus the smaller question of whether it is valid for two
> objects to overlap if one of them is never initialized or
> accessed, since the address range of the implicit extra array
> element may overlap another object.)
>
> The general rule that I would like is:
>
>    Any pointer containing a valid value may be dereferenced.
>    If the resulting lvalue is used in a way which requires a
```

```
        >    complete type, and the pointer does not actually refer to
        >    an object, the behavior is undefined.  [footnote - a
        >    pointer may be valid and yet not refer to an object, e.g. a
        >    pointer to just past the end of an array.]
        >
        > Since this would allow "&*(char*)0", it would require
        > additional wording to prohibit using null pointers this way.
Resolution:
Requestor:      Bill Gibbons
Owner:          Josee Lajoie (Memory Model)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
Work Group:     Core
Issue Number:   721
Title:          Comparisons of pointer to class members need fine tuning
Section:        5.9 [expr.rel]
Status:         editorial
Description:
        5.9/2 says:
          "If two pointers point to nonstatic data members of the same
           object, the pointer to the later declared member compares
           greater provided the two members are not separated by an
           access-specifier label (11.1) and provided their class is not
           a union."

        The "point to" provision probably should also cover "point
        within".

        And the case of pointing just past the end of a member array
        should be mentioned; it is sufficiently difficult to handle
        correctly that I think it is OK just to say that this case is
        unspecified.
Resolution:
Requestor:      Bill Gibbons
Owner:          Josee Lajoie (Object Model)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
Work Group:     Core
Issue Number:   722
Title:          The definition of address constant expression needs fine
                tuning
Section:        5.19 [expr.const]
Status:         editorial
Description:
        5.19/4 address constant expressions
          This needs work.  For example, the phrase "The subscription
          operator ...  can be used" does not describe how it may be
          used; presumably the subscript must be an integral constant
          expression.

        The same goes for 5.19/5.
Resolution:
Requestor:      Bill Gibbons
Owner:          Josee Lajoie (Initialization)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
======================================================================
====
```

Chapter 6 - Statements

==

Chapter 7 - Declarations

Work Group: Core
Issue Number: 683
Title: What is the underlying type of an enumeration type if the value of an enumerator uses the value of a previous enumerator?
Section: 7.2 [dcl.enum]
Status: active
Description:

There is a small omission in the description of the constant-expression which is used to set an enumerator's value, e.g.

```
    enum A { a, b = a + 2 );  // expression "a + 2"
```

The type of "a" in "a+2" presumably follows the usual expression rules. But these rules say, in 4.5/2:

> An rvalue of type wchar_t (3.9.1) or an enumeration type (7.2) can be converted to an rvalue of the first of the following types that can represent all the values of its underlying type: int, unsigned int, long, or unsigned long.

So the evaluation of "a+2" depends on the underlying type of "A", which in turn depends on the value of "b", which depends on the value of "a+2".

Although this is unlikely to affect real programs in practice, we should fix the definition. There are cases where it matters, e.g.:

```
    // Assume an environment where "int" is 16 bits, just for
    // convenience (The same problem occurs when "int" is larger.
    // Think of systems where "int" is 32 bits and "long" is 64
    // bits.)

    enum A { a = 1, b = a-2, c = 32768U };
```

If we assume the underlying type will be "int", then b is -1 and the actual underlying type is "long".

If we assume the underlying type will be "unsigned int", then b is 65535 and the actual underlying type is "unsigned int".

The answer may seem obvious, but consider:

```
    enum A { a = 1U, b = a-2, c = -1 };
```

The underlying type will clearly be signed. Does "b" have the value "-1" or is the code ill-formed?

There seem to be several possible solutions to this problem:

1) When an enumerator is used in the defining expression of a subsequent enumerator in the same enumeration, its type is the type of its defining expression (where the default defining expression is "previous-enumerator + 1" except the first one, where it is "0").

2) Give enumerations an "interim" underlying type which is recomputed after each enumerator, and use that underlying type in subsequent defining expressions.

3) Require that enumerator computation be done with an infinite number of bits - assuming that the "as if" rule makes this practical.

4) Say that if the value of a definining expression depends on the underlying type of the enumeration, the program is ill-formed.

Bill Gibbons' preference is (1).
Bill doesn't think it matters much what the answer is, but the should be described by the working paper.

A related problem occurs with the implicit "next value" rule:

```
enum B { a = 32767, b };
```

Is the code well-formed? If so, what is the underlying type? Why?
This example would be fixed if solution (3) was adopted.

Resolution:
Requestor: Bill Gibbons
Owner: Steve Adamczyk (Types)
Emails: core-6989
Papers:

. . . .

Work Group: Core
Issue Number: 672
Title: using-declarations and base class assignment operators
Section: 7.3.3 [namespace.udecl]
Status: editorial
Description:
7.3.3 should indicate what happens if a using-declaration refers to a base class assignment operator and the type of this assignment operator corresponds to the type of the derived class copy assignment operator.

```
struct B;
struct A {
        B& operator=(const B&);
};
struct B : A {
        // introduces B's copy-assignment operator
        using A::operator=;
};
```

Resolution:
At the Hawaii meeting, members of the core WG wanted the implicit

```
          copy assignment operator for class B still be generated.
          The WP should be clarified to say this.
  Requestor:      Bill Gibbons
  Owner:          Josee Lajoie (Object Model)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
  Work Group:     Core
  Issue Number:   729
  Title:          Must extern "C" functions declared in a namespace and
                  a global extern "C" function have different signatures and
                  return types?
  Section:        7.5 [dcl.link]
  Status:         editorial
  Description:
          extern "C" int f(int);
          namespace NS {
              extern "C" void f(int); // ill-formed? undefined behavior?
          }
  Resolution:
          At the Hawaii meeting, the Core WG agreed that two function
          declarations referring to the same entity must have the same type.
          The case above should be made clearer in the WP.
  Requestor:
  Owner:          Josee Lajoie (extern "C")
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
  Work Group:     Core
  Issue Number:   749
  Title:          Can a declaration specify both a storage class and a
                  linkage specification?
  Section:        7.5[dcl.link]
  Status:         active
  Description:
          What is the meaning of:

             extern "C" static void f();

          Is this still illegal?
          Or does it declare a function with C language linkage that is
          local to the translation unit?

          Mike Anderson proposes the following:
          (1) either the WP should indicate that using a storage class in
              a declaration with a linkage specification with no braces
              is disallowed; or else,

          (2) it should indicate at least that the semantics are
              equivalent whether or not the braces are present and
              possibly do a bit more to specify what the semantics are.

          [Josee:]
            7.5 para 7 says:
            "the form of the linkage-specification directly containing a
             single declaration is treated as an extern specifier for the
             purpose of determining whether the contained declaration is a
             definition.

                extern "C" int i; // declaration
```

"

I believe this implies that the declaration above is equivalent to:

```
    extern static void f();
```

and that Mike's solution (1) is the correct one.

Resolution:
Requestor: Mike Anderson
Owner: Josee Lajoie (extern "C")
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 750
Title: To which declarator in a member function declaration does the extern "C" specifier apply?
Section: 7.5[dcl.link]
Status: editorial
Description:

[Mike Miller in core-7322]:

```
> What is the meaning of 7.5p4, "A non-C++ language linkage is
> ignored ... for the function type of class member function
> declarators" with respect to parameters of member functions?
> For instance,
>
>         extern "C" {
>                 struct S {
>                         void f(void(*)());
>                 };
>         }
>
> Does S::f take a "C" function or a "C++" function?  The
> example in the text deals with related issues but not this
> specific one, and the normative text could be read either way,
> depending on whether you understand "function type of class
> member function declarators" in a shallow or deep sense.
```

[Mike Anderson in core-7323:]

I believe it was intended to be understood in a shallow sense (and that S::f takes a "C" function). The words were crafted to make the rule apply only to certain function types (namely, those of member function declarators) and not to any other function types such as the types of function parameters.

Would it be sufficient to expand the example to make this clear, or does the normative text need to modified? I think another example would be enough.

[Mike Miller in core-7325:]

Assuming that we do intend the "shallow" interpretation, I think the normative words there are wrong; the type of S::f is different ("function taking pointer to C function...") from what it would be if it were not inside extern C ("function taking pointer to C++ function..."), i.e., the non-C++ linkage is *not* ignored in determining the function type. IMHO, it should be rewritten to read something like, "The language linkage of member names and member function types is C++, regardless of the linkage specification in which the class may be defined." (An example is also a good idea.)

Resolution:
Requestor: Mike Miller

```
  Owner:          Josee Lajoie (extern "C")
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
  ========================================================================
   Chapter 8 - Declarators
  ------------------------
  Work Group:     Core
  Issue Number:   689
  Title:          What if two using-declarations refer to the same function but
                  the declarations introduce different default-arguments?
  Section:        8.3.6 [dcl.fct.default]
  Status:         editorial
  Description:
          7.3.3 para 10 says:
          "If the set of declarations and using-declarations for a single
           name are given in a declarative region,
           -- they shall all refer to the same entity, or all refer to
              functions; or ..."

          8.3.6 para 9 says:
          "When a declaration of a function is introduced by way of a using
           declaration, any default argument information associated with the
           declaration is imported as well."

          This is not really clear regarding what happens in the following
          case:
                  namespace A {
                          extern "C" void f(int = 5);
                  }
                  namespace B {
                          extern "C" void f(int = 7);
                  }

                  using A::f;
                  using B::f;

                  f(); // ???
  Resolution:
          At the Hawaii meeting, the core WG agreed that the example above was
          an error and suggested that this be clarified in the WP as an
          editorial matter.
  Requestor:      Bill Gibbons
  Owner:          Josee Lajoie (Default Arguments)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
  Work Group:     Core
  Issue Number:   730
  Title:          When are default arguments for member functions of
                  template classes semantically checked?
  Section:        8.3.6 [dcl.fct.default]
  Status:         active
  Description:
          para 5:
          "The names in the expression are bound and the semantic
           constraints are checked at the point of declaration."
```

```
template<class T> class Cont {
 // ...
public:
 Cont(const T& default_element = T());
 // ...
};

class Y {
public:
 Y(int);
 // ... no Y() ...
};

Cont<Y> y1;  // error: no Y() (that's fine)
Cont<Y> y2(Y(99)); // use 99 as default value
```

However, is the last declaration legal?
When is the checking of the T() for Cont<Y> done?

The current WP implies that it is checked when C<Y> is first instantiated.

If this is the case, all of the standard containers are badly broken - it is not possible to have container with elements of a type without a default constructor.

Bjarne's Proposed Resolution:

The default argument resolution from Stockholm broke the library and should be revised. I suspect that treating a default argument like the return type for an operator->() and the definition of a template member function is the right way (check if and when the default argument is used) and for the same reason: For ordinary classes it makes sense to check when you see the class, for templates that is seriously constraining.

Mike Miller's Proposed Resolution:

The semantic constraints on a default argument should be checked on use, not on declaration, for normal functions as well as template functions. C++ has a number of cases where you can declare things that you cannot use because of unresolvable ambiguities, but we have chosen to diagnose them on use, not on declaration. The rationale for this choice is that diagnosis on declaration prevents composing classes from disparate sources, even though the composition might be useful in ways that do not stumble over the ambiguity.

Mike thinks default arguments are a similar situation -- the function is completely usable as long as you don't rely on the problematic portion of the declaration. While templates are the most likely context in which this issue might arise, I believe there are probably others in non-template situations.

Mike would support a reconsideration of the "immediate diagnosis" part of the Stockholm resolution, preferably altogether, although applying the revision just to templates would still be an improvement.

Resolution:
Requestor: Bjarne Stroustrup
Owner: Steve Adamczyk (Default Arguments)
Emails:

```
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
  Work Group:     Core
  Issue Number:   751
  Title:          Should { } be allowed around an initializer that is a string?
  Section:        8.5[dcl.init]
  Status:         active
  Description:
          The current WP disallows:
             const char a[3] = {"asdf"};
          However, this is allowed in C.

          8.5 paragraph 13 says:
          "If T is a scalar type, then ...
              T x = { a };
           is equivalent to
              T x = a;
          "

          An array is not a scalar type.

          If the committee decides to leave things the way they are, this
          difference between C and C++ should be listed in appendix C.
  Resolution:
  Requestor:
  Owner:          Josee Lajoie (Object Model)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
  ======================================================================
   Chapter 9 - Classes
  --------------------
  Work Group:     Core
  Issue Number:   692
  Title:          ";opt" after member "function-definition" should be omitted
  Section:        9.2 [class.mem]
  Status:         editorial
  Description:
          The syntax says:
          member-declaration:
             ...
             function-definition ;opt

          ";opt" should be omitted. Otherwise, the syntax is ambiguous.
  Resolution:
  Requestor:
  Owner:          (Syntax)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
  Work Group:     Core
  Issue Number:   505
  Title:          Must anonymous unions declared in unnamed namespaces also be
                  declared static?
  Section:        9.5 [class.union] Unions
  Status:         active
  Description:
```

9.5p3 says:
"Anonymous unions declared at namespace scope shall be declared static."
Must anonymous unions declared in unnamed namespaces also be declared static?
If the use of static is deprecated, this doesn't make much sense.

Proposal:
Replace the sentence above with the following:
"Anonymous unions declared in a named namespace or in the global namespace shall be declared static."

This is related to issue 526.

Resolution:
Requestor: Bill Gibbons
Owner: Josee Lajoie (linkage)
Emails:
Papers:

. . . .

==

Chapter 10 - Derived classes

Work Group: Core
Issue Number: 624
Title: class with direct and indirect class of the same type: how can the base class members be referred to?
Sections: 10.1 [class.mi] Multiple base classes
Status: editorial
Description:

para 3 says:
"[Note: a class can be an indirect base class more than once and can be a direct and indirect base class.]"
The WP should describe how base class members can be referred to, how conversion to the base class type is performed, how initialization of these base class subobjects takes place.

Resolution:

At the Stockholm meeting, the core 1 WG decided to handle this as an editorial issue.
A note will be added to the WP to clarify the restrictions on accessing members of the direct base class.

Requestor:
Owner: Josee Lajoie (Object Model)
Emails:
Papers:

. . . .

==

Chapter 11 - Member Access Control

Work Group: Core
Issue Number:a 752
Title: When accessing a base class member, the qualification is not ignored
Section: 11.5[class.protected]
Status: editorial
Description:

11.2 para 4 says:

"The access to a member is affected by the class in which the member is named. This naming class is the class in which the member name was looked up and found. [Note: this class can be explicit, e.g., when a qualified-id is used, or implicit, e.g., when a class member access operator (_expr.ref_) is used (including cases where an implicit this->" is added. If both a class member access operator and a qualified-id are used to name the member (as in p->T::m), the class naming the member is the class named by the nested-name-specifier of the qualified-id (that is, T).]"

This is contradictory to the example in 11.5 para 1:

```
class B {
protected:
    int i;
    static int j;
};

class D1 : public B {
};

class D2 : public B {
    friend void fr(B*,D1*,D2*);
    void mem(B*,D1*);
};
void fr(B* pb, D1* p1, D2* p2)
{
    p2->B::i = 4;  // ok (access through a D2,
                   // *** qualification ignored ***
}
```

According to 11.2 para 4, the qualification is not ignored.

Resolution:
Requestor:
Owner: Steve Adamczyk (Access)
Emails:
Papers:

. . . .
. .

===
====

Chapter 12 - Special Member functions

Work Group: Core
Issue Number: 753
Title: Is 'new char[size]' aligned properly to hold an object of any type T?
Section: 12.4[class.dtor]
Status: active
Description:

[Fergus Henderson in core-7251:]

> The following example in a note in 12.4/13 is not strictly
> conforming C++ according to the rules defined elsewhere in the
> draft. I think it should be changed.
>
> "13[Note: explicit calls of destructors are rarely needed. One
> use of such calls is for objects placed at specific addresses
> using a new- expression with the placement option. Such use
> of explicit placement and destruction of objects can be
> necessary to cope with dedicated hardware resources and for
> writing memory management facilities. For example,
>   void* operator new(size_t, void* p) { return p; }

```
>     struct X {
>         // ...
>         X(int);
>         ~X();
>     };
>     void f(X* p);
>
>     void g()        // rare, specialized use:
>     {
>         char* buf = new char[sizeof(X)];
>         X* p = new(buf) X(222);  // use buf[] and initialize
>         f(p);
>         p->X::~X();              // cleanup
>     }
>  --end note]
> "
>
> The lines
>
>    char* buf = new char[sizeof(X)];
>    X* p = new(buf) X(222);  // use buf[] and initialize
>
> are not strictly conforming, because there is no guarantee
> that `buf' will be sufficiently aligned to hold an object of
> type `X'.  5.3.4[expr.new]/12 includes some examples which
> show that this is not guaranteed.  I think the first of those
> lines should be changed to
>
>        char* bug = ::operator new(sizeof(X));
>
> For stylistic reasons, it might also be a good idea to change
> the line
>
>        p->X::~X();              // cleanup
>
> to  just
>
>        p->~X();
```

[Mike Miller in core-7257:]

```
> Yes, you're right -- there's no requirement that the "array
> allocation overhead" is a multiple of the maximum alignment
> requirement, so the example you cited is not guaranteed to
> work by the current WP text.
>
> However, there's a reason this example is in the WP, and it's
> because this is a very common idiom.  I don't see a compelling
> reason to break it.
>
> I can see three possibilities for accommodating the use of
> "new char[xx]" to get a suitably-aligned buffer space for other
> objects:
> 1) require that the "array allocation overhead" be an
>    integral multiple of the maximum alignment requirement, and
>    that it be required to be a contiguous region between the
>    pointer returned by operator new[] and the pointer to the
>    first element of the array.
> 2) Allow "array allocation overhead" only for arrays of class
>    types (my understanding of the reason for the overhead is
>    to allow the correct invocation of destructors).
> 3) Make char and unsigned char a special case, like they are
>    in many other ways, such that allocating an array of char
>    or unsigned char is guaranteed to have an "array allocation
```

```
          >    overhead" of zero.
          > I guess I don't have a strong preference among the three,
          > although 2 and 3 seem a bit more straightforward and
          > correspond more to the rest of the language.
          >
          > This is obviously not a make-or-break issue; people will
          > continue to write "new char[xx]" and it will continue to work,
          > whether we bless it or not.  But it's not hard to change the
          > WP to allow it, and it would bring us a little closer to
          > reality to recognize this particular practice.
  Resolution:
  Requestor:      Fergus Henderson
  Owner:          Josee Lajoie (Memory Model)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   754
  Title:          for new T, allocation functions in base classes of T
                  are not considered
  Section:        12.5[class.free]
  Status:         editorial
  Description:
          12.5 para 2 says:
          "When a new-expression is used to create an object of class T
           (or array thereof), the allocation function is looked up in the
           scope of class T; if no allocation function is found, the global
           allocation function is used."

          It should be made clearer that allocation functions in base
          classes are not considered.
  Resolution:
  Requestor:      Dan Saks
  Owner:          Josee Lajoie (Memory Model)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   687
  Title:          The WP prohobits the copy assignment of virtual base
classes
                  to behave like the copy constructor
  Section:        12.8 [class.copy]
  Status:         active
  Description:
          The ARM specified:
          "Objects representing virtual base classes will be assigned only
once
           by a generated assignment operator."

          This restriction has been removed.
          The current WP says in 12.8 para 13:
          "The direct base classes of X are assigned first, in the order of
           their declaration in the base-specifier-list, and then the
immediate
           nonstatic data members of X are assigned, in the order in which
           they were declared in the class definition.
           [...]
           It is unspecified whether subobjects representing virtual base
           classes are assigned more than once by the implicitlys-defined
copy
           assignment operator."
```

The new specification does not allow the copy constructor ordering.

Resolution:
Requestor: Bill Gibbons
Owner: Josee Lajoie (Object Model)
Emails:
Papers: 96-0107/N0925

. . . .

Work Group: Core
Issue Number: 755
Title: Assignment of POD class objects: is the class copied as a block?
Section: 12.8[class.copy]
Status: editorial
Description:

[Tom MacDonald compat-353:]

> Recently I became aware of an incompatibility between C and C++
>
> Consider the following example:
>
> ```
> struct S_Pair;
>
> typedef struct Object {
>    struct S_Pair *addr;
>    int tag;
> } Object;
>
> struct S_Pair {
>    Object car;
>    Object cdr;
> };
>
> Object x;
>
> void copy_it(void) {
>
>   x = x.addr->cdr;
>
> }
> ```
>
> The C++ rules permit the following implementation of the structure assignment inside the function copy_it.
>
> ```
>  x.addr = x.addr -> cdr.addr;
>  x.tag  = x.addr -> cdr.tag;
> ```
>
> The C rules are more strict as indicated in 6.3.16.1, the first paragraph under Semantics says:
>
> In simple assignment(=), the value of the right operand is converted to the type of the assignment expression and replaces the value stored in the object designated by the left operand.
>
> Note that the value is spoken of as a whole. There appears to be nothing that allows the identity of the right operand to change in the middle of the assignment, which is the effect what the C++ rules permit.
>
> The second paragraph under Semantics forbids partial overlap. This allows a more efficient implementation of a structure assignment (between lvalues) as

>
>   memcpy(&left_operand, &right_operand)
>
> or an inline equivalent, rather than as
>
>   memmove(&left_operand, &right_operand)
>
> which would include the extra work needed to accommodate the
> possibility of partial overlap (such as copying through a
> temporary object, or deciding whether to copy bytes from the
> beginning or from the end). Note that in either case, the
> addresses of the two operands are computed before the copying
> begins.
>
> The following implementation produces the expected C behavior.
>
> ```
>  {
>  Object * tmp = &(x.addr->cdr);
>  x.addr = tmp->data;
>  x.tag  = tmp->tag;
>  }
> ```

It was not the intention of the C++ standards committee to make
C++ different from C in this case. How could the WP be clarified
to make this intent clearer?

Resolution:
Requestor: Tom MacDonald (C compatibility)
Owner: Josee Lajoie (Memory Model)
Emails:
Papers:

. . . .
. .

==
====

Chapter 13 - Overloading

Work Group: Core
Issue Number: 733
Title: Implicit conversion sequences and scalar types
Section: 13.3.3.1 [over.best.ics]
Status: editorial
Description:

13.3.3.1 para 6:
"The implicit conversion sequence is the one required to convert
the argument expression to an rvalue of the type of the
parameter. ... When the parameter has a class type and the
argument expression is an ravlue of the same type, the implicit
conversion sequence is identity conversion. When a parameter
has class type and the argument expression is an lvalue of the
same type, the implicit conversion sequence is an
lvalue-to-rvalue conversion."

Shouldn't the last two sentences also apply to non-class types?

Jason Merrill also notes in core-7309:

> In this test case, I assert that under the current overloading
> rules the second and third functions are equally good matches
for
> the argument, even though the third is "obviously" the right
> choice. The ics for the third a reference binding to the
lvalue,
> while the ics for the second is a reference binding to a
temporary,

```
> but that also has identity rank because there are no lvalue->rvalue
> conversions for built-in types.  Perhaps there should be?
>
>  int f(char &);
>  int f(const char &);
>  int f(volatile char &);
>  int f(const volatile char &);
>
>  int main()
>  {
>    volatile char c = 'a';
>    f (c);
>  }
```

To which Stephen Adamczyk replies:

```
> I believe there are lvalue-to-rvalue conversions for builtin types.
> Perhaps you're interpreting 13.3.3.1 para 6 (over.best.ics) as
> saying there aren't, because it mentions them explicitly for class
> types but not for builtin types.
> But the class wording is needed because it is a special case. For
> builtin types, the lvalue-to-rvalue conversion is a normal part of
> the implicit conversion sequence, and as 13.3.3.1.1 (over.ics.scs)
> says, that includes an lvalue-to-rvalue conversion when
> appropriate.
```

[Josee:]
I think a note or footnote should be added to make this clear.
I have seen a few compiler writers trip over this.

Resolution:
Requestor:
Owner: Steve Adamczyk (Type Conversions)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 682
Title: operator ?: and operands of enumeration types
Section: 13.6 [over.built]
Status: active
Description:

The type of a conditional expression choosing between two enums of the same type was changed in the May WP from that enum type to the integral type it promotes to, breaking code. I propose changing paragraph 27 of 13.6 [over.built] from

```
27 For every type T, where T is a pointer or pointer-to-member type,
   there exist candidate operator functions of the form
          T       operator?(bool, T, T);
```

to

```
27 For every type T, where T is an enumeration, pointer or
   pointer-to-member type, there exist candidate operator
```

```
functions
             of the form
                    T       operator?(bool, T, T);

          ----------
          Should the following testcase be ambiguous?

            const char c;
            enum E { a } e;
            bool b;

            main ()
            {
              return b ? c : e;
            }

          The builtin candidates are:
              operator ?(bool, const char &, const char &)
              operator ?(bool, int, int)
  Resolution:
  Requestor:      Jason Merrill
  Owner:          Steve Adamczyk (Type Conversions)
  Emails:         core-6983, core-6987
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   734
  Title:          ambiguity in "bool & ? void *& : classType&" where
                  classType has an operator void*&
  Section:        13.6 [over.built]
  Status:         active
  Description:
          This testcase is ambiguous under the current rules:
            void *p;

            struct A {
              operator void*& () { return p; };
            };

            bool b;
            A a;

            main ()
            {
              void *q = b ? p : a;
            }

          The implementation of the current rules results in:
            Ambiguous overload for `bool & ? void *& : A &'
            candidates are: operator ?:(bool, void *&, void *&) <builtin>
                            operator ?:(bool, void *, void *) <builtin>
          because there is no lvalue->rvalue conversion to disambiguate
          for non-class operands.
  Resolution:
  Requestor:      Jason Merrill
  Owner:          Steve Adamczyk (Type Conversions)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   756
  Title:          most uses of built-in "?" with class operands are
```

ambiguous
Section: 13.6[over.built]
Status: active
Description:

The pseudo-prototype for the "?" operator in [over.built] makes most uses of "?" with a class operand ambiguous.

Consider

```
struct A {};
struct B {
  operator A();
};
void f() {
  A a;
  B b;
  1 ? a : b;
}
```

The pseudo-prototype generates the following (and more, but these are enough to demonstrate the ambiguity):

```
bool ? A : A
bool ? const A : const A
```

These are indistinguishable in overload resolution, in the same way that

```
void g(A);
void g(const A);
```

are indistinguishable. As [over.best.ics] para 6 says, in a copy-initialization, "Any difference in top-level cv-qualification is subsumed by the initialization itself and does not constitute a conversion."

Resolution:
Requestor: Steve Adamczyk
Owner: Steve Adamczyk (Type Conversions)
Emails:
Papers:

. . . .

==

Chapter 14 - Templates

Work Group: Core
Issue Number: 757
Title: Can a template member function be overloaded?
Section: 14[temp]
Status: editorial
Description:

14 paragraph 5 says:
"The name of a class template shall not be declared to refer to any other template, class, function, object, enumeration, enumerator, namespace, or type in the same scope (_basic.scope_). Except that a function template can be overloaded either by (non-template) functions with the same name or by other function templates with the same name (_temp.over_), a template name declared in namespace scope shall be unique in that namespace."

```
        This paragraph forgets to say that (except for overloading) the
        name of a function template in class scope must not be the same
        as the name of any other class member.
Resolution:
Requestor:
Owner:          Bill Gibbons (Templates)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
Work Group:     Core
Issue Number:   758
Title:          Can an array name be a template argument?
Section:        14.3[temp.arg]
Status:         editorial
Description:
        14.3[temp.arg] para 3 says:
        "A template-argument for a non-type non-reference
         template-parameter shall be ...  the address of an object or a
         function with external linkage ...  The address of an object or
         function shall be expressed as &f, plain f (for function only)
         ..."

        It is followed by the following example:
          char p[] = "Vivisectionist";
          X<int,p> x2; // & is not used
        i.e. the array name is not preceded with the & operator.

        What was probably intended is the following:
        "The address of an object or function shall be expressed as
         '&e' except when 'e' is a function or an array in which case
         it can be expressed as 'e'."
Resolution:
Requestor:
Owner:          Bill Gibbons (Templates)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
Work Group:     Core
Issue Number:   759
Title:          Initializing a template reference parameter with an
                argument of a derived class type needs to be described
Section:        14.3[temp.arg]
Status:         editorial
Description:
        14.3[temp.arg], paragraph 6:

        "Standard conversions (_conv_) are applied to an expression
         used as a template-argument for a non-type template-parameter
         to bring it to the type of its corresponding
         template-parameter.
         [Example:
            struct Base { /* ... */ };
            struct Derived : Base { /* ... */ };
            template<Base& b> struct Y { /* ... */ };
            Derived d;
            Y<d> yd;   // derived to base conversion
         -- end example]
        "
        Since binding an object of a derived class type to a reference
        to a base class type is not a standard conversion anymore, this
        text needs work.
Resolution:
```

Requestor:
Owner: Bill Gibbons (Templates)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 760
Title: Is a template argument that is a private nested type accessible in the template instantiation context?
Section: 14.3[temp.arg]
Status: active
Description:

Sean Corfield in core-7317:
Is the private nested class accessible in the instantiation context?

```
class Outer {
//...
private:
        class Inner {
        //...
        };
        list< Inner > data;
};
```

Since Outer::Inner is inaccessible outside the scope of Outer and its friends, one can imagine that instantiations would fail. A quick trial on the local compiler agrees (HP's Cfront -- not much of a yardstick).

14.3 [temp.arg] says:
10For a template-argument of class type, the template definition has no special access rights to the inaccessible members of the template argument type. The name of a template-argument shall be accessible at the point where it is used as a template-argument.

All that says is that inaccessible *members* can't be accessed. Is it *really* intending to say that if a template argument is accessible "at the point where it is used as a template-argument" then any & all uses of the corresponding template parameter are accessible within the template body?

```
// Outer::Inner as before
template<typename T>
void A<T>::f() {
      T t; // same as Outer::Inner t but Outer::Inner is not
           // accessible
}
```

I believe we intend that to be well-formed but I just don't think the WP is quite clear enough about it (and certainly some compilers disagree).

Resolution:
Requestor: Sean Corfield
Owner: Bill Gibbons (Templates)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 761
Title: Can the member function of a class template be virtual?

Section: 14.5.1.1[temp.mem.func]
Status: editorial
Description:

14.5.1.1 paragraph 3 says:
"A member function of a class template is implicitly a member function template with the template-parameters of its class template as its template-parameters."
14.5.2 paragraph 3 says:
"A member function template shall not be virtual."

This seems to imply that virtual member functions in a class template are ill-formed.

```
  template <class T> struct AA {
     virtual void f(); // this is an error
  };
```

It should be clarified to say that the following is an error.

```
  template <class T> struct AA {
     template <class C> virtual void f(C); // this is an error
  };
```

We should get rid of the wording in 14.5.1.1 that says that a member function of a class template is a member function template with the template parameters of its class. This sentence is confusing.

Resolution:
Requestor:
Owner: Bill Gibbons (Templates)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 762
Title: How can function template be overloaded?
Section: 14.5.5.1[temp.arg]
Status: editorial
Description:

14.5.5.1 para 4 says:
"The signature of a function template consists of its function signature, its return type and its template parameter list. The names of the template parameters are significant only for establishing the relationship between the template parameters and the rest of the signature."

I think an example showing that two function templates that have the same function parameter list are valid overloads would make it clear that such thing is allowed. For example:

```
template<class T> void f();
template<int I> void f(); // valid overload
```

Resolution:
Requestor:
Owner: Bill Gibbons (Templates)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 763
Title: Partial Specialization: the transformation also affects the function return type
Section: 14.5.5.2[temp.func.order]
Status: editorial

Description:

14.5.5.2 [temp.func.order] paragraph 2 says:
"The transformation used is:

- -- For each type template parameter, synthesize a unique type and substitute that for each occurrence of that parameter in the function parameter list.
- -- For each non-type template parameter, synthesize a unique value of the appropriate type and substitute that for each occurrence of that parameter in the function parameter list."

These bullets should say:
"... in the function parameter list _and return type_".

Resolution:
Requestor:
Owner: Bill Gibbons (Templates)
Emails:
Papers:

. . . .
. .

Work Group: Core
Issue Number: 736
Title: How can/must typename be used?
Section: 14.6 [temp.res]
Status: active
Description:

Is typename required in situations where we know only type names can be used?

```
   class typename T::X var; // or class T::X var; ?
```

Other situations:

```
  o base class names
  o before ::
  o operator typename T::X or operator T::X ?
  o dynamic_cast<typename T::X> or dynamic_cast<T::X> ?
```

What if typename is used preceding a template dependent name that is not qualified? Is typename ignored, or is this ill-formed?

```
template <class T> class C {
  typename C<T> ...
};
```

What if typename is used preceding an non-dependant name? Is typename ignored, or is this ill-formed?

```
class A { };
template <class T> class C {
  typename A ...
};
```

Is the following well formed?

```
template<typename T, typename typename T::X R>
   class A { };
```

It is not totally clear how typename can be used in a template parameter list.

The WP needs to be clearer about these cases.

Resolution:
Requestor:
Owner: Bill Gibbons/John Spicer (Templates)
Emails:
Papers:

. . . .
. .

```
  Work Group:     Core
  Issue Number:   764
  Title:          undeclared name in template definition should be an error
  Section:        14.6[temp.names]
  Status:         editorial
  Description:
          The example in 14.6 paragraph 1 has the following lines:

            T::A* a7;// T::A is not a type name:
            // multiply T::A by a7
            B* a8;   // B is not a type name:
            // multiply B by a8; ill-formed,
            // no visible declaration of B

          The first line is also ill-formed because a7 is not declared.
  Resolution:
  Requestor:
  Owner:          Bill Gibbons (Templates)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   765
  Title:          The syntax does not allow the keyword 'template' in
                  'expr.template C<parm>'
  Section:        14.6[temp.names]
  Status:         editorial
  Description:
          In 14.2[temp.names], paragraph 4 says:

          "When the name of a member template specialization appears
           after .  or -> in a postfix-expression, or after :: in a
           qualified-id that explicitly depends on a template-argument
           (_temp.dep_), the member template name must be prefixed by the
           keyword template.  Otherwise the name is assumed to name a
           non-template."

          The grammar in 14.6 paragraph 2 does not seem to take this into
          account:

          elaborated-type-specifier:
            . . .
            typename ::(opt) nested-name-specifier identifier
            typename ::(opt) nested-name-specifier identifier
                                            < template-argument-list >

          shouldn't this say?

          elaborated-type-specifier:
            . . .
            typename ::(opt) nested-name-specifier template(opt) identifier
            typename ::(opt) nested-name-specifier template(opt) identifier
                                            < template-argument-list >
  Resolution:
  Requestor:
  Owner:          Bill Gibbons (Templates)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   766
  Title:          How do template parameter names interfere with names in
```

```
                nested namespace definitions?
Section:        14.6.1[temp.local]
Status:         active
Description:
        14.6.1[temp.local] paragraph 6 says:
          "In the definition of a member of a class template that
           appears outside of the class template definition, the name
           of a member of this template hides the name of a
           template-parameter.
           [Example:
              template<class T> struct A {
                      struct B { /* ... */ };
                      void f();
              };

              template<class B> void A<B>::f()
              {
                      B b;  // A's B, not the template parameter
              }
           -- end example]
          "

        This does not cover namespaces very well.
        For example, what happens when a template parameter names
        conflicts with the name of a namespace member.

            namespace N {
                    struct B { /* ... */ };
                    template<class T> void f(T);
            }
            template<class B> void N::f(B)
            {
                    B b;  // A's B or the template parameter?
            }

        John Spicer's proposed resolution:
          You should get the same result whether the function is
          defined in the class (or namespace) or outside of it.
          The "B" in N::f gets the template parameter B, not the
          namespace member B.
Resolution:
Requestor:
Owner:          Bill Gibbons (Templates)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
Work Group:     Core
Issue Number:   737
Title:          How can dependant names be used in member declarations
                that appear outside of the class template definition?
Section:        14.6.4 [temp.dep.res]
Status:         editorial
Description:
         template <class T> class Foo {
           public:
           typedef int Bar;
           Bar f();
         };
         template <class T> typename Foo<T>::Bar Foo<T>::f() { return 1;}
                            --------------------

         In the class template definition, the declaration of the member
         function is interpreted as:
```

```
  int Foo<T>::f();
```

In the definition of the member function that appears outside of the class template, the return type is not known until the member function is instantiated. Must the return type of the member function be known when this out-of-line definition is seen (in which case the definition above is ill-formed)? Or is it OK to wait until the member function is instantiated to see if the type of the return type matches the return type in the class template definition (in which case the definition above is well-formed)?

From John Spicer:

```
> My opinion (which I think matches several posted on the
> reflector recently) is that the out-of-class definition must
> match the declaration in the template.  In your example they
> do match, so it is well formed.
>
> I've added some additional cases that illustrate cases that
> I think either are allowed or should be allowed, and some
> cases that I don't think are allowed.
>
> template <class T> class A { typedef int X; };
>
> template <class T> class Foo {
> public:
>   typedef int Bar;
>   typedef typename A<T>::X X;
>   Bar f();
>   int g1();
>   Bar g2();
>   X h();
>   X i();
>   int j();
> };
>
> // Declarations that are okay
> template <class T> typename Foo<T>::Bar Foo<T>::f()
>                                                 { return 1;}
> template <class T> typename Foo<T>::Bar Foo<T>::g1()
>                                                 { return 1;}
> template <class T> int Foo<T>::g2() { return 1;}
> template <class T> typename Foo<T>::X Foo<T>::h() { return 1;}
>
> // Declarations that are not okay
> template <class T> int Foo<T>::i() { return 1;}
> template <class T> typename Foo<T>::X Foo<T>::j() { return 1;}
>
> In general, if you can match the declarations up using only
> information from the template, then the declaration is valid.
>
> Declarations like Foo::i and Foo::j are invalid because for
> a given instance of A<T>, A<T>::X may not actually be int if
> the class is specialized.
>
> This is not a problem for Foo::g1 and Foo::g2 because for
> any instance of Foo<T> that is generated from the template
> you know that Bar will always be int. If an instance of Foo
> is specialized, the template member definitions are not used
> so it doesn't matter whether a specialization defines Bar as
> int or not.
```

Resolution:

Core 3 agreed that this is largely editorial.

Some work is needed to figure out exactly what needs to be said.
Owner: Bill Gibbons/John Spicer (Templates)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 767
Title: Where should the point of instantiation of class templates be discussed?
Section: 14.6.4.1[temp.point]
Status: editorial
Description:

14.6.4.1[temp.point]:
Shouldn't this subclause also discuss the point of instantiation of class templates?

14.7.1 covers some aspect of the point of instantiation of class templates.

Having a subclause called "point of instantiation" and only discuss function templates within it is somewhat confusing.

Resolution:
Requestor:
Owner: Bill Gibbons (Templates)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 677
Title: Should the text on argument deduction be moved to a subclause discussing both function templates and class template partial specializations?
Section: 14.8.2 [temp.deduct]
Status: editorial
Description:

Template argument deduction is now used both for function templates and for class template partial specializations. The text for temp.deduct should be moved out of the function template specializations subclause.

Here is the reorganization Bill Gibbons suggested in private email:

```
> 14.2 Names of template specializations (including functions)
> 14.3 Template arguments (including functions; cross-ref arg
>      deduction)
> ...
> 14.8  Template argument deduction
> 14.8.1  Deducing a template argument from an expression
> 14.8.2  Argument deduction for function calls
> 14.8.3  Argument deduction for partial specialization ordering
>
> 14.9  Function calls
> 14.9.1  Mixing explicit and deduced template arguments
> 14.9.2  Overload resolution
> 14.9.3  Overloading and template specializations
```

Resolution:
Requestor: Sean Corfield
Owner: Bill Gibbons/John Spicer (Templates)
Emails:

```
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   768
  Title:          typename keyword missing in some examples
  Section:        14.8.2[temp.deduct]
  Status:         editorial
  Description:
          14.8.2 paragraph 10 is an error

            template<int i, typename T>
              T deduce(A<T>::X x,     // T is not deduced here
                       T       t,     // but T is deduced here
                       B<i>::Y y);    // i is not deduced here
              A<int> a;
              B<77>  b;
              int    x = deduce<77>(a.xm, 62, y.ym);
              // T is deduced to be int, a.xm must be convertible to
              // A<int>::X
              // i is explicitly specified to be 77, y.ym must be
convertible
              // to B<77>::Y

          According to 14.6 paragraph 2
          "A qualified-name that refers to a type and that depends on a
           template-parameter shall be prefixed by the keyword typename"

          A<T>::X x above should be: typename A<T>::X x
          B<i>::Y y above should be: typename B<i>::Y y
  Resolution:
  Requestor:
  Owner:          Bill Gibbons (Templates)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  ========================================================================
====
   Chapter 15 - Exception Handling
  ---------------------------------
  Work Group:     Core
  Issue Number:   769
  Title:          Are the base class dtors called if the derived dtor
                  throws an exception?
  Section:        15.2[except.dtor]
  Status:         active
  Description:
          [Mike Ball, core-7288:]

                 #include <iostream.h>

                 struct base{
                   ~base() { cerr << "base\\n";}
                 };

                 struct derived : public base{
                   ~derived() { throw("error"); }
                 };

                 void doit() {
                   derived x;
                 }
```

```
int main() {
  try {
    doit();
  } catch(...) {
  }
  return 0;
}
```

Should the destructor for "base" be executed? The answer is not in the DWP, though it does state that it will be executed if the destructor for "derived" has a function catch block.

I would consider this an obvious editorial matter were it not that I can think of reasons that the programmer might want the base class destructors not to be executed. For example, there is otherwise no way to abort a destructor in the middle. The current specification provides a way to achieve that. The programmer could have the base destructors executed by providing a function catch block and have them skipped by not providing one.

This is pretty thin reasoning, but it implies that this is not so obvious.

[Jerry Schwarz, core-7289:]

I assume that the destructor for the base class wouldn't be called.

To clarify my reasoning: the calling of the base subobject's destructor is part of the execution of the derived class constructor, and it wouldn't be executed any more than would statements following the throw. And I'll note that the same question might be asked about the member subobjects. For which I assume the answer would be the same. (Whatever that is.)

[Bjarne, core-7290:]

It has been a principle throughout that constructed sub-objects are destroyed if a constructor throws an exception. Consider a base an unnamed member and it all works out.

[John Skaller, core-7294:]

I assume the base destructor IS called.

There are TWO reasons to destroy the object, the first is that the user code invoked the destructor, and the second is that the exception requires object/stack unwinding.

Even if the exception is somehow caught, that still leaves the program to continue destroying the object normally.

The only way the destruction can be stopped is by calling a special handler, terminate() or perhaps unexpected().

[Erwin Unruh, core-7297:]

My opinion is that a compound statement can be seen as a corner case of a try statement which just has no handler. In this light I would argue to have the same semantics with a compound statement than with a handler whose catch clauses don't match.

This would argue in calling the base destructors. This would not allow base destructors to be avoided. But if a programmer wants this, he can put a flag into the base object and have the destructor check this flag. So the restriction is not too hard.

Current practice:
[Anthony Scian, core-7299:]
I tried the program under Watcom C++, MS VC++, and Borland C++ with the result that all three C++ implementations destructed the base class.

Resolution:
Requestor: Mike Ball
Owner: Bill Gibbons (Exception Handling)
Emails:
Papers:

. . . .

==

Chapter 16 - Preprocessing Directives

==

Annex C - Compatibility

Work Group: Core
Issue Number: 680
Title: Annex C subclause C.1 is out of date
Section: C.1 [diff.c]
Status: editorial
Description:

Jonathan Schilling wrote the following:

The introduction to Annex C (Compatibility) and subclause C.1 (Extensions) both look like they were quickly edited from the base document for use in the standard, but the edit missed some spots and left others making no sense ("... from the dialects of Classic C used up till now", "... since the 1985 version of this manual"). More attention is given to Classic C than is now necessary, and the new features list is very incomplete.

The proposed rewrite of the introduction and subclause C.1 is below.

An alternative course of action would be to drop C.1 altogether, but I think that once made accurate it serves a useful purpose.

Proposed Resolution:

Replaced C.1 and C.1.1 with:

Annex C (informative)
Compatibility [diff]

This Annex summarizes the evolution of C++ and explains in detail the differences between C++ and ISO C, both in the language and in the standard library.

With the exceptions listed in this Annex, programs that are both C++ and C have the same meaning in both languages. All differences between C++ and C can be diagnosed by an implementation, although converting programs between C++ and C may be subject to the vicissitudes of unspecified and undefined behavior.

C.1 Extensions [diff.c]

This subclause summarizes the major extensions to C provided by C++. Because C++ was originally based upon the C of the first edition of _The C Programming Language_, before C became an ISO standard, there was some parallel evolution between the two languages. This is noted here by the phrase "also in ISO C".

C.1.1 C++ features available in 1985 [diff.early]

This subclause summarizes the extensions to C provided by C++ by 1985, as described in the first edition of _The C++ Programming Language_:

< same feature list that's in current [diff.early] >

C.1.2 C++ features added 1985 - 1991 [diff.mid]

This subclause summarizes the major extensions to C++ between 1985 and 1991, as described in the second edition of _The C++ Programming Language_:

< same feature list that's in current [diff.c++], except:
take out "The bool type" (20)
take out the references to things being "moved to the anachronism subclause" (5, 8) >

C.1.3 C++ features added since 1991 [diff.late]

This subclause summarizes the major extensions to C++ since 1991, as described in this International Standard:

Universal character names ([lex.charset]), trigraphs ([lex.trigraph]), and operator keywords ([lex.key]).

The bool type; [basic.fundamental].

The wchar_t type; [basic.fundamental].

User-defined new and delete operators for arrays; [expr.new], [expr.delete].

Placement delete; [expr.new].

Run-time type identification, including dynamic_cast and typeid; [expr.dynamic.cast], [expr.typeid].

A new form for casts: static_cast ([expr.static.cast]), reinterpret_cast ([expr.reinterpret.cast]), and const_cast ([expr.const.cast]).

Declarations in tested conditions in if, switch, for, and while statements; [stmt.select], [stmt.iter].

Namespaces; [basic.namespace].

Class members can be declared mutable; [decl.stc].

The explicit keyword for providing non-converting constructors; [dcl.fct.spec].

Forward declaration of nested classes; [class.nest].

Static data member constants; [class.static.data].

Relaxation of the rule for return types of overriding functions; [class.virtual].

Overloading based on enumerations; [over.load].

Refinement of the template compilation model and addition of the export keyword; [temp].

The typename keyword in template parameters; [temp.param].

Default arguments for template type parameters; [temp.param].

Default arguments for template type parameters; [temp.param].

Explicit template argument specification in template function calls; [temp.arg.explicit].

Explicit template instantiation; [temp.explicit].

New syntax for template specialization; [temp.expl.spec].

Partial specialization of class templates; [temp.class.spec].

Member templates; [temp.mem].

Function try blocks; [except].

The uncaught_exception() function; [except.uncaught].

The C++ Standard library; [lib.library].

Resolution:

Requestor: Jonathan Schilling

Owner: Tom Plum (C compatibility)

Emails:

compat-352

Papers:

. . . .

Work Group: Core

Issue Number: 743

Title: Some anachronisms are missing from annex C

Section: C.3 [diff.anac]

Status: editorial

Description:

Annex C (Compatibility), subclause C.3 (Anachronisms), seems very odd as it stands. It covers only the oldest and probably least-used anachronisms supported by compilers. Only some of them relate to use of C programs as C++.

A more current list would include lots of other things, such as anachronisms due to Cfront 3.0 peculiarities, anachronisms due to differences between the ARM and the WP, and so on (see the anachronism list for any commercial compiler for how long these can get, e.g. EDG).

Jonathan proposes to reduce subclause C.3 to a single paragraph providing for anachronism support in general, without any specific items. The proposed wording:

C.3 Anachronisms [diff.anac]

Extensions to the C++ language may be provided by an

```
          implementation to ease the use of C programs as C++ programs or
          to provide continuity from earlier C++ implementations.  Note
          that use of such extensions is likely to have undesirable
          aspects.  An implementation providing them should also provide a
          way for the user to ensure that they do not occur in a source
          file.  A C++ implementation is not obliged to provide these
          features.
  Resolution:
          At the Hawaii meeting, the C compatibility WG decided that annex
          C.3 should either be removed or rewritten.
  Requestor:      Jonathan Schilling
  Owner:          Tom Plum (C compatibility)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  ==========================================================================
====
   Annex E - Universal-character-names
  ------------------------------------
  Work Group:     Core
  Issue Number:   770
  Title:          The title of Annex E needs to be made shorter
  Section:        Annex E[extendid]
  Status:         editorial
  Description:
          The top of page E-2 (Annex E) has the section title overlapping
          the date.

          Andrew Koenig responded the following:
          > The reason is that (major) clause titles aren't checked for
          > overlap with the date.  The easiest fix is therefore to
          > rename clause E to something shorter.
  Resolution:
  Requestor:
  Owner:          Tom Plum (Annex E)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  ==========================================================================
====
  +--------------+
  | Closed Issues |
  +--------------+

  The following issues were resolved at the Hawaii meeting.
  These issues
    o were addressed by motions adopted at the Hawaii meeting, or
    o were editorial issues corrected by editorial actions at the Hawaii
      meeting, or
    o were rejected because the Core WG decided to take no action.

  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .

  1.8 [intro.execution]:
    605: The execution model wrt to sequence points and side-effects needs
work
    693: What can be done within a signal handler?
  2.1 [lex.phases]:
    695: A source file must not end in a new-line character - is a
diagnostic
```

```
         required?
  2.8 [lex.header]:
    696: What happens if // appears between < and >?
    697: Should special characters in include file names be
         implementation-defined?
  2.13 [lex.ccon]:
    698: Should wide-character literals be well-defined for a given locale?
    699: What is the size of a non-wide string literal that contains UCNs?
  3.2 [basic.def.odr]:
    700: Is a diagnostic required if a function or object is not defined?
  3.3.1 [basic.scope.pdecl]:
    701: Is a class type first used in the parameter list of a function
         definition introduced in the function outermost block?
  3.4.2 [basic.lookup.koenig]:
    702: When do "member functions" hide functions from associated
namespaces?
    703: What are the associated namespaces of a template-id?
  3.4.4 [basic.lookup.elab]:
    666: Are class names used in an elaborated-type-specifier hidden by
         namespace names?
  3.4.5 [basic.lookup.classref]:
    704: e.B::a, must B be an unambiguous base class of e's class?
  3.5 [basic.link]:
    526: What is the linkage of names declared in unnamed namespaces?
    705: What is the linkage of non-exported templates?
    706: extern block scope declarations and lookup of previous "matching"
         declarations
  3.6.3 [basic.start.term]:
    663: Should the meaning of a coexisting C/C++ implementation be
defined?
  3.7.3 [basic.stc.dynamic]:
    667: What does "predeclared" operator new mean?
    707: Implications of the predeclared operator new(size_t)
  3.7.3.1 [basic.stc.dynamic.allocation]:
    708: What can a user-declared allocation do if it fails to allocate
         storage?
  3.9 [basic.types]:
    709: Can one use memcpy to copy the content of objects of non-POD type?
  3.10 [basic.lval]:
    710: A union with a char array should alias with other types
  4.2 [conv.array]:
    668: Should the conversion from string-literal to pointer to char be an
         "array-to-pointer" conversion which has exact match rank in
function
         overload resolution?
  5.2.4 [expr.pseudo]:
    715: cv-qualifiers and pseudo destructor calls
  5.2.7 [expr.dynamic.cast]:
    549: Is a dynamic_cast from a private base allowed?
  5.2.8 [expr.typeid]:
    716: Can a class type be defined in a typeid expression?
  5.2.10 [expr.reinterpret.cast]:
    717: Can a static_cast cast an incomplete class type to its own type?
  5.3.4 [expr.new]:
    638: When is access/ambiguity on operator delete checked?
  5.5 [expr.mptr.oper]:
    644: Must the operand of .* and ->* have a complete class type?
  5.9 [expr.rel]:
    670: Is the comparison between void* and cv T* well-formed?
  5.17 [expr.ass]:
    691: is bool += 1 valid?
  5.19 [expr.const]:
    723: Should pointer to member casts be allowed in pointer to member
         constant expressions?
  6.2 [stmt.expr]:
    645b: When is the result of an expression statement converted to an
rvalue?
  6.4.2 [stmt.switch]:
    724: Should the integral constant-expression be converted to the
promoted
          type of the switch condition?
  6.7 [stmt.dcl]:
    635: local static variable initialization and recursive function calls
    725: Can a local object be initialized before the first time control
passes
          through its declaration?
  6.8 [stmt.ambig]:
    671: Does template instantiation happen during parser ambiguity
resolution?
  7.1.2 [dcl.fct.spec]:
    726: inline functions must be declared inline in all translation units
-
          is a diagnostic required?
  7.3.1.2 [namespace.memdef]:
    727: In which namespace are names in extern block declarations and
          function block declarations looked up?
  7.3.3 [namespace.udecl]:
    673: Does a using-declaration for an enum type declare aliases for the
          enumerator names as well?
  7.3.4 [namespace.udir]:
    612: name look up and unnamed namespaces
  7.5 [dcl.link]:
    728: How are extern "C" objects declared or defined?
  10.2 [class.member.lookup]:
    674: How do using-declarations affect class member lookup?
  10.3 [class.virtual]:
    675: How do using-declarations influence the selection of a final
virtual
          function overrider?
  11.4 [class.friend]:
    731: Do functions first declared as friends still have external
linkage?
  12.3 [class.conv]:
    732: Should "explicit" be allowed on type conversion operators?
  12.6 [class.init]:
    138: When are default ctor default args evaluated for array elements?
  13.3.1.1.2 [over.call.object]:
    662: Do cv-qualifiers on the class object influence the operator()
called?
  13.3.3.2 [over.ics.rank]:
    684: The ranking for implicit conversion sequences for pointer types
should
          take into account qualification conversions in 4.4
    685: What is the ranking of a user-defined conversion that combines a
          pointer conversion with casting away cv-qualifiers?
  14.1 [temp.param]:
    735: Semantics for some forms of the template parameter missing?
  14.6 [temp.res]:
    738: Can a template parameter be declared as a friend?
  14.7.1 [templ.inst]:
    676: When is a template instantiated?
  14.8.2 [temp.deduct]:
    739: How does argument deduction works if operator T is a member
template?
  15.1 [except.throw]:
    678: Can the exception object created by a throw expression have array
  15.4 [except.spec]:
    740: Can an exception specification appear in a reference declaration?
  15.5.3 [except.uncaught]:
    741: The definition of uncaught_exception does not take into account
         nested exceptions
  clause 16:
    679: "Shall" is used incorrectly in clause 16
    742: Should __STDC__ be in the list of predefined macros?
  Annex C:
    681: The type of string literals is array of const char - this has
         implicitions for C compatibility and should be in Annex C
  ===========================================================================
====
   Chapter 1 - Introduction
  -------------------------
  Work Group:     Core
  Issue Number:   605
  Title:          The execution model wrt to sequence points and side-
effects
                  needs work
  Section:        1.8 [intro.execution]
  Status:         closed
  Description:
          See UK issues 263, 264, 265, 266:
          1.8 para 9:
          "What is a "needed side-effect"? This paragraph, along with
           footnote 3 appears to be a definition of the C standard "as-if"
           rule.  This rule should be defined as such.  [Proposed
definition
           of "needed": if the output of the program depends on it.]"

          Bill Gibbons also notes:
          > [1.8/1] The "as-if" rule seems too important to leave as a
          > footnote.  I suggest promoting it to normative text in 1.3 or
          > expanding 1.8/1.  We should probably name this rule so it can
          > be more easily referenced.

          1.8 para 10:
          "It is not true to say that values of objects at the previous
           sequence point may be relied on.  If an object has a new value
           assigned to it and is not of type sig_atomic_t the bytes making
up
           that object may be individually assigned values at any point
prior
           to the next sequence point.   So the value of any object that is
           modified between two sequence points is indeterminate between
those
           two points.  This paragraph needs to be modified to reflect this
           state of affairs."

          Also, para 11:
          "Such an object [of automatic storage duration] exits and retains
its
           last-stored value during the execution of the block and while
the
           block is suspended ..."
          This is not quite correct, the object may not retain its last-
stored
          value.

          Para 9, 10, 11 and 12 also contain some undefined terms.
  Resolution:
          A definition for the as-if rule has been provided.
          Paragraph 10 was substantially reworked.
  Requestor:      UK issues 263, 264, 265, 266
  Owner:          Steve Adamczyk (sequence points)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   693
  Title:          What can be done within a signal handler?
  Section:        1.8 [intro.execution]
  Status:         closed
  Description:
          [1.8/10]:
          "When the processing of the abstract machine is interrupted by
           receipt of a signal, only the values of objects as of the
           previous sequence point may be relied on.  Objects that may be
           modified between the previous sequence point and the next
           sequence point need not have received their correct values yet."

          Shouldn't it also say that if the handler modifies any variable
          which is also modified between the sequence points then the
          value of the variable becomes undefined?

          Erwin Unruh adds:
          > In C there is a big restriction of what you can do inside a
          > signal handler.  You cannot call any library function (with 3
          > exceptions) and you may not access or modify any global
          > variable (except with type 'volatile sig_atomic_t').
          >
          > In C++ we have inherited the signal function.  So we have to
          > check what restrictions are needed in C++.  Regarding the
          > common subset of C and C++ we can adopt the rules of C.
          >
          > Some very basic C++ constructs are critical. Two examples:
          >
          > -- Constructing a class object may put the address of the vtbl
          >    into the object.  The equivalent code would not be strictly
          >    conforming in C.
          >
          > -- Declaring a variable with a destructor.  In usual code
          >    this needs some adjustment so that the destructor will be
          >    called when an exception is encountered.  In a portable
          >    implementation this would be done by pushing a description
          >    object on a global stack.
          >
          > So I would like to have a rule along the lines of:
          >
          > A function registered as a signal handler may only do what it
          > is entitled to do in the C standard.  A function which uses
          > (even potentially) a language or library feature not in C will
          > cause undefined behaviour.
          >
          > [Note: This also covers very minor additions!
          > [Example:
          >
          >  inline void f(){}  // inline is no C
          >  void g(int) { if (0) f(); } // g uses a non-C feature
          >
          >  signal( SIGINT, &g );       // undefined behaviour
          > ]
          > Although f is never called, activating a SIGINT causes
          > undefined behaviour.  Note that using exception handling or
          > RTTI would most probably cause problems on some machines.]
          >
          > I know this rule is overly restrictive.  On the other hand
          > trying to figure out what really is possible inside a signal
```

```
        > handler will need too much time.  In C the rule is: The only
        > thing you can portably do is setting a global flag.  My rule
        > will keep that rule and allow an implementation to mostly
        > ignore the possibility of signals.
  Resolution:
        Paragraph 10 has been modified to say:
        "When the processing of the abstract machine is interrupted by
         receipt of a signal, the values of objects modified after the
         preceding sequence point are indeterminate during the execution
of
         the signal handler, and the value of any object not of
         volatile sig_atomic_t that is modified by the handler becomes
         undefined."
  Requestor:      Bill Gibbons & Erwin Unruh
  Owner:          (Execution Model)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  ========================================================================
====
   Chapter 2 - Lexical Conventions
  ---------------------------------
  Work Group:     Core
  Issue Number:   695
  Title:          A source file must not end in a new-line character -
                  is a diagnostic required?
  Section:        2.1 [lex.phases]
  Status:         closed
  Description:
        2.1p1: "A source file that is not empty shall end in a new-line
                character"

        Should this be "no diagnostic required?"
        Current implementations vary in this regard.

        [Mike Miller:]
        Is there a reason for the rule in the first place?  Why should
        a compiler care whether I hit the Return key in my editor before
        saving the buffer to disk for compilation?

        [Josee:]
        This text is taken directly from C.
        In C a diagnostic is required.
  Resolution:
        2.1 p1 now says:
        "If a source file that is not empty does not end in a new-line
         character, or ends in a new-line character immediately preceded
by
         a backlash character, the behavior is undefined."
  Requestor:      Mike Miller
  Owner:          Tom Plum (Lexical Analysis)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   696
  Title:          What happens if // appears between < and >?
  Section:        2.8 [lex.header]
  Status:         closed
  Description:
        2.8 para2:
        "If the characters ...  /* appear in the sequence between the <
```

and > delimiters ... the the behavior is undefined."

Should this also include "//"?

[Josee:]
I believe the // were omitted by mistake when the text was copied from the C standard. I also believe this is an editorial matter.

Resolution:
The WP text was modified such that if the // are found, the behavior is also undefined.

Requestor: Mike Miller
Owner: Tom Plum (Lexical Analysis)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 697
Title: Should special characters in include file names be implementation-defined?
Section: 2.8 [lex.header]
Status: closed
Description:
[2.8/2]:
"If the characters ', \, ", or /* appear in the sequence between the < and > delimiters, or between the " delimiters, the behavior is undefined."

Why not implementation-defined?

Resolution:
The C compatibility WG decided to leave this the way it was: the behavior is undefined.

Requestor: Bill Gibbons
Owner: Tom Plum (Lexical Analysis)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 698
Title: Should wide-character literals be well-defined for a given locale?
Section: 2.13.2 [lex.ccon]
Status: closed
Description:
[2.13.2]:
"Wide-character literals have implementation-defined values, regardless of the number of characters in the literal"

Why do wide-character literals have implementation-defined values? Shouldn't they have the value specified by the execution character set? (Which may be locale-dependent, but at least is well-defined for a given locale.)

Resolution:
The WP has been modified according to the suggestion above and now says:
"The value of a wide-character literal containing a single c-char has value equal to the numerical value of the encoding of the c-char in the execution wide-character set. The value of a wide-character literal containing multiple c-chars is implementation-defined."

```
  Requestor:      Bill Gibbons
  Owner:          Tom Plum (Lexical Analysis)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   699
  Title:          What is the size of a non-wide string literal that
                  contains UCNs?
  Section:        2.13 [lex.ccon]
  Status:         closed
  Description:
          [2.13.4/5]
          "The size of a non-wide string literal is the total number of
           escape sequences and other characters, plus at least one for
           the multibyte encoding of each universal-character-name"

          This needs to be improved.

          * I thought the UCN proposal said that UCN's which were not
            representable in the execution character set were to be
            mapped to some single character in the execution character
            set.  This would preclude multibyte encodings.  (The wording
            from N0886 is "A universal-character-name is translated to the
            encoding, in the execution character set, of the character
            named.  If there is no such encoding, the
            universal-character-name is translated to an
            implementation-defined encoding." I take this as meaning
            "implementation-defined encoding, in the execution character
            set" which I interpret as encoding in a single character.  Was
            this not the intent, or was it changed?)

          * If a UCN is representable in the execution character set, its
            multibyte encoding is a single byte so the "plus one" is
            wrong.

          * The term "multibyte encoding" is not defined, although
            "multibyte character" is.  I suggest something like "plus at
            least one for each universal-character-name which is not
            representable in the execution character set and which the
            implementation translates into a multibyte character
            appropriately encoded."
  Resolution:
          The compatibility WG decided to leave the things as they were.
  Requestor:      Bill Gibbons
  Owner:          Tom Plum (Lexical Analysis)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  ==========================================================================
===
   Chapter 3 - Basic Concepts
  ----------------------------
  Work Group:     Core
  Issue Number:   700
  Title:          Is a diagnostic required if a function or object is not
                  defined?
  Section:        3.2 [basic.def.odr]
  Status:         closed
  Description:
          "Every program shall contain at least one definition of every
           function used in that program. ... An object that is used in
```

a program shall be defined."

Should this say: No diagnostic required. ?
Is the answer different for virtual functions that are neither called nor used to form a pointer to member?

[Josee:]
If diagnostics are supposed to be issued to help users identify portions of their code that may not be portable from one implementation to another, isn't the WP correct requiring a diagnostic in all these cases?

Resolution:
This portion of the text now says: no diagnostic required.

Requestor: Mike Miller
Owner: Josee Lajoie (ODR)
Emails:
Papers: 96-0174/N0992

. . . .

Work Group: Core
Issue Number: 701
Title: Is a class type first used in the parameter list of a function definition introduced in the function outermost block?
Section: 3.3.1 [basic.scope.pdecl]
Status: closed
Description:

3.3.1/5 says:
"for an elaborated-type-specifier of the form
class-key identifier
the identifier is declared as a class-name in the smallest non-class, non-function prototype scope that contains the declaration."

This implies that for:

```
void f(struct A *a);
void g(struct B *b) { }
```

the name "A" is inserted in the scope outside the function, while the name "B" is inserted in the outermost block of "g", since that is the scope of parameter declarations in a function definition.

3.3.1p6 should be changed to declare the identifier in the scope containing the function definition, not in the outermost block of the function definition.

Resolution:
3.3.1 para 5 was modified to properly cover Bill's example above:
"for an elaborated-type-specifier of the form
class-key identifier
if the elaborated-type-specifier is used in the decl-specifier-seq or parameter-declaration-clause of a function defined in namespace scope, the identifier is declared as a class-name in the namespace that contains the declaration; otherwise, except as a friend declaration, the identifier is declared in the smallest non-class, non-function prototype scope that contains the declaration."

Requestor: Bill Gibbons
Owner: Josee Lajoie (Name Lookup)
Emails:

```
   Papers:
   . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
   Work Group:     Core
   Issue Number:   702
   Title:          When do "member functions" hide functions from associated
                   namespaces?
   Section:        3.4.2 [basic.lookup.koenig]
   Status:         closed
   Description:
           3.4.2 [basic.lookup.koenig] paragraph 2 says:
             "If the ordinary unqualified lookup of the name finds the
              declaration of a member function, the associated namespaces
              are not considered."

           Does `member function' mean `member of class' or could `member
           of namespace' be considered.  If the latter, is the global
           namespace considered.  Here is an example:

           namespace A {
               struct S { ... };
               void f(A::S);
               void g(A::S);
           }

           void f(A::S); // member of ::
           void g(A::S); // member of ::

           namespace C {
               void f(A::S); // member of C

               void h()
               {
                   A::S a;
                   f(a); // C::f, ::f, A::f, or ambiguous?
                   g(a); // ::g, A::g, or ambiguous?
               }
           }
   Resolution:
           3.4.2 para 2 was modified to say:
           "If the ordinary unqualified lookup of the name finds the
            declaration of a _class_ member function, ..."
   Requestor:      Bjarne Stroustrup
   Owner:          Josee Lajoie (Name Lookup)
   Emails:
   Papers:
   . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
   Work Group:     Core
   Issue Number:   703
   Title:          What are the associated namespaces of a template-id?
   Section:        3.4.2 [basic.lookup.koenig]
   Status:         closed
   Description:
           3.4.2/2 says:
           "If T is a template-id, its associated namespaces are the
            namespace of the template and the namespaces associated with
            the type of template arguments."

           Bill Gibbons:
             Should anything be said about non-type arguments?  I suggest
             that for *value* non-type arguments, there are no associated
             namespaces.  For *linkage-name* non-type arguments (i.e. those
             where the specialization is based on the name of some entity
```

with external linkage), the associated namespace could be the namespace of the argument.

Mike Miller asked:
How about if the value non-type argument is an enumerator? Shouldn't that have the associated namespace of the enumeration?

Resolution:
3.4.2/2 has been modified to say:
"If T is a template-id, its associated namespaces are the namespace in which the template is defined, the namespaces associated with the types of template arguments provided for template type parameters (excluding template template parameters), and the namespace of any template template arguments. [Note: non-type template arguments do not contribute to the set of associated namespaces.]"

Requestor: Bill Gibbons
Owner: Josee Lajoie (Name Lookup)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 666
Title: Are class names used in an elaborated-type-specifier hidden by namespace names?
Section: 3.4.4 [basic.lookup.elab]
Status: closed
Description:
3.4.4 para 1:
"An elaborated-type-specifier may be used to refer to a previously declared class-name or enum-name even though the name has been hidden by an object, function, or enumerator declaration."

Shouldn't this list also include namespace names?

```
struct S { };
namespace A {
    namespace S {
        struct S sb; // ill-formed? or does it find ::S?
    }
}
```

Resolution:
The sentence above was modified as follows:
"... even though the name has been hidden by non-type declaration."

In the example above, S refers to ::S.

Requestor:
Owner: Josee Lajoie (Name Lookup)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 704
Title: e.B::a, must B be an unambiguous base class of e's class?
Section: 3.4.5 [basic.lookup.classref]
Status: closed
Description:
A a

```
 /   \
 B   B
 |   |
 C   D
 \   /
   E

 E e;
 e.B::a
   -
```

Is this well-formed, or should the WP say that B is a ambiguous base class of e's class and hence the expression above is ill-formed?

```
   A a
 /   \
 B   E

 E e;
 e.B::a
   -
```

If the above OK even if B is not a base class of E?

Whatever the outcome, this should be made clearer.

Resolution:

The WP has been clarified to say:

[Note: the result of looking up the class-name-or-namespace-name is not required to be a unique base class of the class type of the object expression, as long as the entity or entities named by the qualified-id are members of the class type of the object expression and are not ambiguous according to _class.member.lookup_.

```
struct A {
        int a;
};
struct B: virtual A { };
struct C: B { };
struct D: B { };
struct E: public C, public D { };
struct F: public A { };
void f() {
        E e;
        e.B::a = 0;     // ok, only one A::a in E

        F f;
        f.B::a = 1;     // ok, A::a is a member of F
}
```

--end note]

Requestor:
Owner: Josee Lajoie (Name Lookup)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 526
Title: What is the linkage of names declared in unnamed namespaces?
Section: 3.5 [basic.link] Program and linkage
Status: closed

Description:

What is the linkage of names declared in an unnamed namespace?
Internal linkage?
Internal linkage applies to variables and functions.
What would the status of a type definition be in an unnamed namespace? No linkage?
Can it be used to declare a function with external linkage?
Can it be used to instantiate a template?

```
namespace {
  class A { /* ... */ };
}
extern void f(A&);                             // error?
template <class T> class X { /* ... */ };
X<A> x;                                        // error?
```

If A does not have external linkage, then the two declarations are probably errors. If it does have external linkage, then the two declarations are legal (and the implementation probably has to worry about name mangling).

Resolution:

The current rules indicate that the linkage of entities declared in a namespace is external linkage. At the Hawaii meeting, the members of the core WG decided that this applies for entities declared in unnamed namespaces as well.
i.e. leave things the way they are.

Requestor: Mike Anderson

Owner: Josee Lajoie (Linkage)

Emails: core-5905 and following messages.

Papers:

. . . .

Work Group: Core

Issue Number: 705

Title: What is the linkage of non-exported templates?

Section: 3.5 [basic.link]

Status: closed

Description:

Linkage is not the sole determinant of whether identifiers in different translation units can potentially refer to the same entity. For templates, whether they are "export" or not is also a factor (a non-export template definition cannot be referenced outside its translation unit, even if it has external linkage). 3.5 says that even non-export function templates have external linkage unless explicitly declared static. Either 3p8 needs to be rewritten to mention the "export" status of templates, or the definition of linkage needs to change to say that non-export templates have internal linkage.

Resolution:

At the Hawaii meeting, the core WG decided to leave things as they are: all templates have external linkage.

Requestor: Mike Miller

Owner: Josee Lajoie (Linkage)

Emails:

Papers:

. . . .

Work Group: Core

Issue Number: 706

Title: extern block scope declarations and lookup of previous "matching" declarations

Section: 3.5 [basic.link]
Status: closed
Description:

3.5/6 contains the example and text:

```
static void f();
static int i = 0; //1
void g() {
    extern void f(); // internal linkage
    int i; //2: 'i' has no linkage
    {
        extern void f(); // internal linkage
        extern int i; //3: external linkage
    }
}
```

"If the block scope declaration matches a prior visible declaration of the same object, the name introduced by the block scope declaration receives the linkage of the previous declaration; otherwise, it receives external linkage."

Bill Gibbons:

I think the wording is too subtle. He think of "match" as meaning "same name" and possibly "same type". Apparently here it means "same storage duration" too.

And you get into trouble with ambiguities; what about:

```
namespace A { extern int x; }
namespace B { static float x; }
void f() {
    using namespace A;
    using namespace B;
    extern int x;
}
```

Is "x" extern because it matched "A::x" but not "B::x"? What if "B::x" had been type "int"; does that make the example ill-formed?

Resolution:

3.5 para 6 was clarified as follows:

"The name of a function declared in block scope, and the name of an object declared by a block scope extern declaration, have linkage. If there is a visible declaration of an entity with linkage having the same name and type, ignoring entities declared outside the innermost enclosing namespace scope, the block scope declaration declares that same entity and receives the linkage of the previous declaration. If there is more than one such matching entity, the program is ill-formed. Otherwise, if no matching entity is found, the block scope entity receives external linkage."

Bill's example is therefore ill-formed because there exists two visible entities for the extern declaration `extern int x;'.

Requestor: Mike Miller
Owner: Josee Lajoie (Linkage)
Emails:
Papers:

. . . .

```
  Work Group:     Core
  Issue Number:   663
  Title:          Should the meaning of a coexisting C/C++ implementation
be
                  defined?
  Section:        3.6.3 [basic.start.term]
  Status:         closed
  Description:
          3.6.3 Termination [basic.start.term], paragraph 4 states:
            "Where a C++ implementation coexists with a C implementation,
             any actions specified by the C implementation to take place
             after the atexit functions have been called take place after
             all destructors have been called."

          What exactly does it mean for a C++ implementation to "coexist"
          with a C implementation?

          Is this quoted paragraph a constraint on conforming C++
          implementations?  That would raise the spectre where a C++
          implementation could be rendered non-conforming by the mere
          *existence* of a certain (perhaps maliciously designed) C
          implementation!

          Is the quoted paragraph a constraint on C implementations?
          (But how could this be?  How could the C++ standard constrain C
          implementations, which don't claim to conform to the C++
          standard?)

          Or is the quoted paragraph simply a non-normative "hint" to
          compiler writers, the sort of thing that John Skaller would
          probably call meaningless waffle?  (In which case, what is it
          doing in the main text of the standard?)

          As the draft currently stands, I believe the third alternative
          is the most reasonable interpretation, although frankly the
          draft is not clear.
  Resolution:
          The paragraph in question was deleted.
  Requestor:      Fergus Henderson
  Owner:          Josee Lajoie (Memory Model)
  Emails:         core-6823
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   667
  Title:          What does "predeclared" operator new mean?
  Section:        3.7.3 [basic.stc.dynamic]
  Status:         closed
  Description:
          3.7.3 para 2 says:
          "The following allocation and deallocation functions are
implicitly
           declared in a program
                  ::operator new(size_t)
                  ::operator new[](size_t)
                  ::operator delete(void*)
                  ::operator delete[](void*)
          "

          One implication of having predeclared operators is that the
          declarations would have to be explicitly repeated if there were
other
          overloads of operator new declared in global scope, otherwise the
```

overload declarations would hide the implicit declaration. For instance,

```
void* operator new(size_t, long); // hides predeclared op new

int* i = new int;       // ill-formed: no operator new(size_t)
                        // visible at this point
```

It seems that it depends on how we define "implicitly declared" to work -- are "implicit declarations" considered to be in an imaginary scope containing the global scope, or are implicit declarations in the global scope itself and act just the way an explicit declaration would in the global scope? Is it well-defined somewhere what "implicitly declared" means? We need to pin it down.

Resolution:

3.7.3 has been clarified as follows:

"The library provides default definitions for the global allocation and deallocation functions. Some global allocation and deallocation functions are replaceable (_lib.new.delete_). A C++ program shall provide at most one definition of a replaceable allocation or deallocation function. Any such function definition replaces the default version provided in the library (_lib.replacement.functions_). The following allocation and deallocation functions (_lib.support.dynamic_) are implicitly declared in global scope in each translation unit of a program

```
  void* operator new(std::size_t) throw(std::bad_alloc);
  void* operator new[](std::size_t) throw(std::bad_alloc);
  void operator delete(void*) throw();
  void operator delete[](void*) throw();
```

"

Because the declarations appear in global scope, additional user declared operator new functions will overload the predeclared ones.

Requestor: Mike Miller
Owner: Josee Lajoie (Memory Model)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 707
Title: Implications of the predeclared operator new(size_t)
Section: 3.7.3 [basic.stc.dynamic]
Status: closed
Description:

para 2:

"The following allocation/deallocation functions are implicitly declared in a program:

```
   ...
   void *operator new(size_t)
```

"

-- Editorially, should 3.7.3 be changed to include the appropriate exception-specifications?

-- Does this imply that namespace std is predefined?

```
               Is the following ill-formed?

               int std;
               int main() { }

            ---------------------
            Also:
            15.4p2 says, "If any declaration of a function has an
            exception-specification, all declarations, including the
            definition and an explicit specialization, of that function
            shall have an exception-specification with the same set of
            type-ids."

            -- Is it required that declarations and definitions of a
               user-supplied replacement operator new(size_t) have an
               exception-specification naming (exactly) std::bad_alloc, by
               virtue of the predeclared status of operator new(size_t),
               even if <new> is not #included anywhere in the program?

            The resolution for these issues should be made explicit one way
            or the other.
    Resolution:
            To answer the first question:
            3.7.3 has been clarified as follows:
            "These implicit declarations introduce only the function names
             operator new, operator new[], operator delete, operator
delete[].
             [Note: the implicit declarations do not introduce the names std,
             std::bad_alloc, and std::size_t, or any other names that the
library
             uses to declare these names. Thus, a new-expression,
             delete-expression or function call that refers to one of these
             functions without including the header <new> is well-formed.
             However, referring to std, std::bad_alloc, and std::size_t is
             ill-formed unless the name has been declared by including the
             appropriate header."

            To answer the second question:
            3.7.3.1 para 3 was modified as follows:
            "An allocation function that fails to allocate storage can invoke
the
             currently installed new_handler (_lib.new.handler_).  [Note: A
             program-supplied allocation function can obtain the address of
the
             currently installed new_handler using the set_new_handler
function
             (_lib.set.new.handler_).] If a nothrow allocation function
             (_lib.support.dynamic_) fails to allocate storage, it shall
return
             a null pointer. Any other allocation function that fails to
allocate
             storage shall only indicate failure by throwing an exception of
             class std::bad_alloc (_lib.bad.alloc_) or a class derived from
             std::bad_alloc."
    Requestor:      Mike Miller
    Owner:          Josee Lajoie (Memory Model)
    Emails:
    Papers:
    . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
    Work Group:     Core
    Issue Number:   708
    Title:          What can a user-declared allocation do if it fails to
                    allocate storage?
```

Section: 3.7.3.1 [basic.stc.dynamic.allocation]
Status: closed
Description:
3.7.3.1 para 3:
"If an allocation function is unable to obtain an appropriate block of storage, it can invoke the currently installed new_handler and/or throw an exception of class bad_alloc or a class derived from bad_alloc."

Is this supposed to be an exhaustive list of responses to allocation failure? Can an allocation function return 0 or a distinguished value? Does it have to use the new_handler in a specified fashion (e.g., retry after return)? There's more that needs to be said here, I think.

[Josee:]
According to my understanding, the answers to Mike's questions are:
yes, no, yes.
Clarifications need to make this more explicit.
Resolution:
See the resolution for the previous issue.
Requestor: Mike Miller
Owner: Josee Lajoie (Memory Model)
Emails:
Papers:
. . . .
Work Group: Core
Issue Number: 709
Title: Can one use memcpy to copy the content of objects of non-POD type?
Section: 3.9 [basic.types]
Status: closed
Description:
para 2:
"For any object type T, whether or not the object holds a valid value of type T, the underlying bytes making up the object can be copied into an array of char or unsigned char. If the content of the array of char or unsigned char is copied back into the object, the object shall subsequently hold its original value."

1.7p4 only guarantees contiguity for POD types. Doesn't this provision assume and require contiguity for all types?

Shouldn't para 2 only apply to objects of POD types?
Resolution:
The wording above was modified so that the rule only applies to POD
objects.
Requestor: Mike Miller
Owner: Josee Lajoie (Memory Model)
Emails:
Papers:
. . . .
Work Group: Core
Issue Number: 710
Title: A union with a char array should alias with other types
Section: 3.10 [basic.lval]
Status: closed
Description:
Mike suggests:

The "char or unsigned char" bullet should be moved before the "aggregate or union type" bullet; otherwise, a union with a char array would not be able to alias other types, even though pointers and references to char and unsigned char are able to do so.

```
int i = 13;
union A { int ui; char a[sizeof(int)]; };
union B { char a[sizeof(int)]; };
A* ap = reinterpret_cast<A*>(&i);
B* bp = reinterpret_cast(B*){&i);
ap->a[n] = ...; // This is okay
bp->a[n] = ...; // This is undefined behavior
```

Josee:
Is the above really valid?
In C, the "character type" bullet comes last.

Mike Miller"
All the other bullets deal with types that can be "overlaid" onto an object (presumably via pointer or reference cast). For example,

```
int i;
struct B { };
struct D:B { } d;
void f() {
  // The following are all defined behavior because of
  // the referenced bullets in 3.10p14:

  i;    // bullet 1
  *((const int*) &i);  // bullet 2
  *((unsigned*) &i);  // bullet 3
  *((const unsigned*) &i); // bullet 4
  *((B*) &d);   // bullet 6
  *((char*) &i);   // bullet 7
}
```

It therefore seems reasonable to interpret bullet 5 likewise:

```
union U { int j; char c;};
void g() {
  // The following are also defined behavior:

  *((U*) &i);   // bullet 5
  ((U*) &i)->j;   // bullets 5 and 1
  ((U*) &i)->c;   // bullets 5 and 7
}
```

Resolution:
The core WG decided at the Hawaii meeting to leave things the way they are.

Requestor: Mike Miller
Owner: Josee Lajoie (Object Model)
Emails:
Papers:

. . . .

==

Chapter 4 - Standard Conversions

Work Group: Core
Issue Number: 668
Title: Should the conversion from string-literal to pointer to

```
char
                  be an "array-to-pointer" conversion which has exact match
                  rank in function overload resolution?
  Section:        4.2 [conv.array]
  Status:         closed
  Description:
          4.2 para 2:
          "A string literal ... can be converted to an rvalue of type
           "pointer to char"... the result is a pointer to the first
element of
           the array."

          The conversion of a string literal from the type "const char *"
to
          the type "char *" is in the array-to-pointer conversion section.
          This means that this conversion is ranked as an exact match
during
          function overload resolution.  i.e.

                  void f(char*);
                  void f(const char*);
                  f("abc"); // ambiguous

          When the conversion is eventually removed (it is currently
          deprecated), then the call above will be well-formed, and
          void f(const char*) will be chosen. This is different from Kevlin
          Henney's proposal, which suggested that the function
          void f(const char*) be selected.

          In private email, Steve Adamczyk noted that core 2 didn't notice
the
          impact of the proposed wording on the overload resolution
weighting.
  Resolution:
          The call above will prefer f(const char *).

          4.2 para 2 now says:
            "For the purpose of ranking in overload resolution
             (_over.ics.scs_), this conversion is considered an
             array-to-pointer conversion followed by a qualification
             conversion (_conv.qual_).  [Example: "abc" is converted to
             "pointer to const char" as an  array-to-pointer  conversion,
              and then to "pointer to char" as a qualification conversion."
  Requestor:
  Owner:          Steve Adamczyk (Type Conversions)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  ==========================================================================
====
   Chapter 5 - Expressions
  -----------------------
  Work Group:     Core
  Issue Number:   715
  Title:          cv-qualifiers and pseudo destructor calls
  Section:        5.2.4 [expr.pseudo]
  Status:         closed
  Description:
          5.2.4/2 discusses pseudo destructor calls
          "The type designated by the pseudo-destructor-name shall be the
           same as the object type."

          Should a type that only has different cv-qualifiers be allowed?
```

i.e.

```
    const int x;
    x.~int();     // "const int" != "int"
```

or:

```
    typedef const int CI;
    int y;
    y.~CI();      // "int" != "const int"
```

I have no recommendation here, but I think the WP should say something about these cases.

Resolution:

For two types to be the same, they must have the same cv-qualifiers.

At the Hawaii meeting, the core WG decided that the current limitation was acceptable.

Requestor: Bill Gibbons
Owner: Josee Lajoie (Object Model)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 549
Title: Is a dynamic_cast from a private base allowed?
Section: 5.2.7 [expr.dynamic.cast]
Status: closed
Description:

paragraph 8 says:

"...if the type of the complete object has an unambiguous public base class of type T, the result is a pointer (reference) to the T sub-object of the complete object. Otherwise, the runtime check fails."

This contradicts the example that follows:

```
class A { };
class B { };
class D : public virtual A, private B { };
...
D d;
B* bp = (B*) &d;
D& dr = dynamic_cast<D&>(*bp); // succeeds
```

According to the wording in paragraph 8, the cast above should fail.

Bill Gibbons noted the following:

First, the access restrictions on dynamic_casts appear to come from the access restrictions on static_cast, where neither upcasting nor downcasting across private derivation is allowed.

Yet dynamic_cast does not apply these restrictions consistently, even for simple downcasts:

```
        struct A { virtual void f() { } };
        struct B : private A { };
        struct C : public  B { };
        void f() {
            A *a = (A*) new C;
            B *b = static_cast<B*>(a);  // ill-formed
            B *b = dynamic_cast<B*>(a); // OK under 1st "otherwise"
        }
```

I see several ways to clean this up:

(1) Change the first "otherwise" clause to also require that "v points (refers) to a public base class sub-object of the most derived object". This seems closest to the intent of the current wording. It would make the above example ill-formed.

This is equivalent to saying that a dynamic cast is OK if it can be done with a static cast to the most derived type followed by a static cast to the final type, ignoring the uniqueness and virtual inheritance restrictions on static downcasts.

(2) Say something like:

A dynamic cast is well-formed if there exists a class X within the most derived object hierarchy (including the most derived class) such that:

-- "v" refers to X or a public base class of X; and

-- T is X or a public base class of X.

That is, a dynamic cast is OK if it can be done with any combination of two static casts, ignoring the uniqueness and virtual inheritance restrictions on static downcasts. This would also make the above example ill-formed.

(3) Change both dynamic_cast and static_cast; see below.

I had also forgotten (and was somewhat dismayed to rediscover) that static_cast cannot be used to break protection. For example:

```
        struct A { };
        struct B : private A { };
        void f() {
            B *b = new B;
            A *a1 = (A*) b;              // OK
            A *a2 = static_cast<A*>(b);  // ill-formed
            A *a3 = dynamic_cast<A*>(b); // well-formed,
                                         // but "a3" not usable
        }
```

Did we really intend to do this, or was it an accidental side effect of defining static_cast in terms of the inverse of an implicit cast?

Also, I see no reason to restrict downcasting across private inheritance. If static_cast were changed to allow it, I would consider the "across private inheritance" part to be implicit, and the "downcasting" part to be the one that required an explicit cast.

In that light, I would propose one of these changes to dynamic_cast:

(1) Remove the first "public" from paragraph 8 and also allow downcasting to the most derived class, regardless of access.

(2) The equivalent of (2) above:

A dynamic cast is well-formed if there exists a class X within the most derived object hierarchy (including the most derived class) such that:

-- "v" refers to X or a base class of X; and

-- T is X or a public base class of X.

That is, a dynamic cast is OK if it can be done with a combination of two static casts, ignoring the uniqueness and virtual inheritance restrictions on static downcasts. This would also make the above example ill-formed.

Similarly, should upcasting of pointers to members across private inheritance be restricted more than upcasting of pointers to members across public inheritance?

Resolution:

The description of the semantics of the dynamic_cast were clarified as follows:

"8 The run-time check logically executes as follows:

--If, in the most derived object pointed (referred) to by v, v points (refers) to a public base class sub-object of a T object, and if only one object of type T is derived from the sub-object pointed (referred) to by v, the result is a pointer (an lvalue referring) to that T object.

--Otherwise, if v points (refers) to a public base class sub-object of the most derived object, and the type of the most derived object has an unambiguous public base class of type T, the result is a pointer (an lvalue referring) to the T sub-object of the most derived object.

--Otherwise, the run-time check fails.

[Example:

```
        class A { virtual void f(); };
        class B { virtual void g(); };
        class D : public virtual A, private B {};
        void g()
        {
           D   d;
           B*  bp = (B*)&d;  // cast needed to break protection
           A*  ap = &d;      // public derivation, no cast needed
           D&  dr = dynamic_cast<D&>(*bp);  // fails
           ap = dynamic_cast<A*>(bp);       // fails
           bp = dynamic_cast<B*>(ap);       // fails
           ap = dynamic_cast<A*>(&dr);      // succeeds
           bp = dynamic_cast<B*>(&dr);      // fails
        }
     "
```

Requestor:
Owner: Bill Gibbons (RTTI)
Emails:
Papers:

. . . .
. .

Work Group: Core
Issue Number: 716
Title: Can a class type be defined in a typeid expression?
Section: 5.2.8 [expr.typeid]
Status: closed
Description:

Steve Clamage:
The following article appeared in comp.std.c++. The Sept draft in 5.2.8 does not prohibit defining a type in a typeid expression, but also doesn't say what the meaning is if you do so.

In article 6ke@news.service.uci.edu, dan@cafws4.eng.uci.edu (Dan Harkless) writes:

> The Draft Standard explicitly says that you can't define a
> type in a sizeof expression or inside a cast. However, it
> says nothing about defining types within a typeid expression.
> Does this mean it's allowed,or is here something somewhere
> else making it illegal and the sections for sizeof and the
> casts are just being redundant?
>
> If it is legal, does the type get defined within the scope
> the typeid expression appears in, or is it just alive for the
> purposes of making type_info object?

Resolution:

The WP now says:
"Types shall not be defined in the type-id."

Requestor: Steve Clamage
Owner: Bill Gibbons (RTTI)
Emails:
Papers:

. . . .
. .

Work Group: Core
Issue Number: 717
Title: Can a static_cast cast an incomplete class type to its own type?
Section: 5.2.9 [expr.static.cast]
Status: closed
Description:

5.2.9/1 says about static_cast:
"[in static_cast<T>(v)]... T shall not be an incomplete class type, a pointer to an incomplete class type, or a reference to

an incomplete class type. v shall not be a pointer to an incomplete class type, or an lvalue that has incomplete class type."

This prohibits:

```
struct T;
void f(T *pt, void *pv) {
    pt = static_cast<T*>(pt);    // identity conv. not allowed
    pv = static_cast<void*>(pt); // cast to void* not allowed
    pt = static_cast<T*>(pv);    // cast from void* not allowed
}
```

Is this intentional?

Resolution:
The above two sentences were deleted.

Requestor: Bill Gibbons
Owner: Steve Adamczyk (Type Conversions)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 638
Title: When is access/ambiguity on operator delete checked?
Section: 5.3.4 [expr.new] New
Status: closed
Description:

5.3.4 para 15 indicates that access and ambiguity on operator delete are checked when a new expression is encountered.

This does not seem quite right for objects of class type with a virtual destructor.

Some tricky examples were provided on the reflector during the discussion on this topic:

Example 1:

Roly Perera [core-6993]:

```
> struct B {
>     virtual ~B ();
>     void operator delete (void*);
> };
>
> struct D : B {
>     void operator delete (void*);
> };
>
> int main () {
>     B* pb = ::new D; // 1.  requires ::delete
>     delete pb;       // 2.  should find D::operator delete
> }
```

The deallocation function used by the delete expression could be the class operator delete even if the new expression uses global operator new. So the ambiguity/access of the class operator delete should always be checked.

Example 2.

Erwin Unruh [core-6997]:

```
> struct B {
>     virtual ~B ();
>     void operator delete (void*);
> };
>
> struct D : B {
>     void operator delete (void*) { /* does nothing !! */ }
> };
>
> int main () {
>     D d;
>     pb = &d;
>     delete pb;
>     exit(1);
> }
```

Erwin's example (though somewhat sick ;-) shows that a delete expression can be used without any new operator ever being called to create the object. The example deletes a local variable and since the operator delete does nothing, only the destructor is run.
The destructor at the end of the block is bypassed by the call to exit. (yuck!).

Erwin says:

> I am perfectly happy to make this program ill-formed. But I as an implementor would like to have a rule which makes sure that I never try to call an operator delete [at runtime] which is ambiguous or inaccessible. Having undefined behaviour is a bad solution.

Resolution:

12.4 now says in paragraph 11:
"At the point of definition of a virtual destructor (including an implicit definition (_class.copy_)), non-placement operator delete shall be looked up in the scope of the destructor's class (_basic.lookup.unqual_) and if found shall be accessible and unambiguous. [Note: this assures that an operator delete corresponding to the dynamic type of an object is available for the delete-expression (_class.free_).]"

Requestor: John Skaller
Owner: Josee Lajoie (Memory Model)
Emails: core-6988
Papers:

. . . .

Work Group: Core
Issue Number: 644
Title: Must the operand of .* and ->* have a complete class type?
Section: 5.5 [expr.mptr.oper]
Status: closed
Description:

Para 2:
"The binary operator .* binds its second operand, which shall be of type ``pointer to member of T '' to its first operand, which shall be of class T or of a class of which T is an unambiguous and accessible base class."

And something similar in para 3 for the ->* operator.

Since pointer to members of an incomplete class type are allowed, i.e.

8.3.3 para 2 says:

```
"   class T;
    char T::* pmc;
 [...]
 the declaration of pmc is well-formed even though T is an incomplete
 type."
```

Must T be a complete class type when a pointer to member operator .* or ->* is applied to the pointer to member?

Resolution:
5.5. now requires that the pointer to member be a pointer to member to a complete class T before it can be the operand of the .* or ->* operator.

Requestor: Jerry Schwarz
Owner: Bill Gibbons (Pointer to members)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 670
Title: Is the comparison between void* and cv T* well-formed?
Section: 5.9 [expr.rel]
Status: closed
Description:

5.9 para 2
"Pointer conversions and qualification conversions are performed on pointer operands ... to bring them to the same type, which shall be a cv-qualified or cv-unqualified version of the type of one of the operands."

Should the following be well-formed?

```
const int * pci;
void * pv;

pv == pci; // well-formed?
```

The current wording indicates that it is ill-formed since the common type of the operands, after pointer conversions and qualification conversions are applied, is 'const void *'.
The wording says that the type to which both operands are converted
"shall be a cv-qualified or cv-unqualified version of the type of one of the operands."

According to 3.9.3 paragraph 1, the cv-qualified versions of 'void *' is 'void * const', 'void * volatile' or 'void * const volatile'. Because 'const void *' is not a cv-qualified version of 'void *', the comparison above is ill-formed.

However, the code above is valid C code.

Either the comparison above should be well-formed (in which case the wording that says: "which shall be a cv-qualified or cv-unqualified version of the type of one of the operands" needs to

be fixed) or, it is ill-formed (in which case annex C needs to
indicate this incompatibility between C and C++).

5.16[expr.cond] has similar problems.

The note that follows says:
[Note: this implies that any pointer can be compared to a null
pointer constant and that any object pointer can be compared to a
pointer of cv-qualified or cv-unqualified type void* (in the latter
case the pointer is first implicitly converted to void*).]

The part about "can be compared to a pointer of cv-qualified or
cv-unqualified type void*" is not quite true, since you can't do:

```
    void f(const int *p, volatile void *q) {
        p < q;
    }
```

since neither "p" nor "q" can be converted to the other's type.

Is the following well formed?

```
    struct A { };
    struct B : A { };
    struct C : A { };
    void f(B *b, C *c) {
        b < c;
    }
```

Bill Gibbons think they should be.

Resolution:
See 96-0125/N1033 in the post-Hawaii mailing.

Requestor: Bill Gibbons
Owner: Steve Adamczyk (Type Conversions)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 691
Title: is bool += 1 valid?
Section: 5.17 [expr.ass]
Status: closed
Description:

5.17 para 7:
"The behavior of an expression of the form E1 op= E2 is equivalent to
E1 = E1 op E2 except that E1 is evaluated only once. In += and -=,
E1 shall either have arithmetic or enumeration type or be a pointer
to a possibly cv-qualified completely defined object type. In all
other cases, E1 shall have arithmetic type."

Can E1 have type bool? If yes, what are the semantics?

Resolution:
Yes, E1 can have type bool.
The result of this expression is already covered by the conversions
from bool to int and from int to bool in clause 4.

Requestor:

```
  Owner:          Steve Adamczyk (Type Conversions)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   723
  Title:          Should pointer to member casts be allowed in pointer to
                  member constant expressions?
  Section:        5.19 [expr.const]
  Status:         closed
  Description:
          5.19/6 pointer to member constant expressions
          I don't see any reason to disallow pointer to member casts here.
  Resolution:
          5.19 para 6 now reads:
          "A pointer to member constant expression shall be created using
the
           unary & operator applied to a qualified-id operand
           (_expr.unary.op_), optionally preceded by a pointer to member
cast
           (_expr.static.cast_)."
  Requestor:      Bill Gibbons
  Owner:          Josee Lajoie (Initialization)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  ====================================================================
====
   Chapter 6 - Statements
  ----------------------
  Work Group:     Core
  Issue Number:   645b
  Title:          When is the result of an expression statement converted
to an
                  rvalue?
  Section:        6.2 [stmt.expr]
  Status:         closed
  Description:
          class C;
          extern C& f();
          void foo() {
                  f(); //1
          }

          Is line //1 ill-formed because the return value of f() is
converted
          to an rvalue and C is an incomplete class type?
  Resolution:
          See 96-0215/N1033 in the post-Hawaii mailing.
  Requestor:
  Owner:          Steve Adamczyk (Type Conversions)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   724
  Title:          Should the integral constant-expression be converted to
                  the promoted type of the switch condition?
  Section:        6.4.2 [stmt.switch]
  Status:         closed
  Description:
```

6.4.2/2 says about case labels in switch statements:
"The integral constant-expression (5.19) is implicitly
converted to the promoted type of the switch condition."

This produces less robust behavior than one might want. Consider the following somewhat contrived example, written for a machine with 32-bit int and 64-bit long:

```
const unsigned long op1 = 0;
const unsigned long op2 = 429467296UL; // 2^32
template<class T> void anyAction(T t) {
    switch (t) {
    case op1:
        // ...
    case op2:
        // ...
    }
}
void smallAction(unsigned x) {
    anyAction(x);
}
```

This is ill-formed because when anyAction<unsigned> is instantiated, the type of "t" is "unsigned int" so "op1" and "op2" are converted to "unsigned int", and the converted values are both zero. (Duplicate case labels are not allowed.)

I think the above example should be well-formed. I can think of two simple ways to do that:

* The case labels are not converted at all, and each comparison of the switch value to a case label is done using the usual rules for "==". This can be optimized to be just as efficient as the current behavior, but it works in a natural and obvious way for all switch values an labels (unlike the current rules).

* Determine a comparison type in a manner similar to the "usual arithmetic conversion" rules, and convert both the switch value and the case labels to that type before comparing.

Both methods allow a jump-table implementation, and for the vast majority of cases have the same semantics and implementation. I believe the only changes in semantics are when a narrowing conversion implied by the current rules is not value-preserving, and this case is almost certainly a bug in the program anyway.

Resolution:
At the Hawaii meeting, the core WG decided to leave things the way they are.

Requestor: Bill Gibbons
Owner: Steve Adamczyk (Type Conversions)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 635
Title: local static variable initialization and recursive function calls
Section: 6.7 [stmt.dcl]
Status: closed
Description:

```
int foo(int i) {
        if (i == 0) return i;
        static int x ( foo (i-1) );
        return x;
}
... foo (10) ...
```

What is the value of x after it has been initialized?

The WP indicates that the variable "x" will be initialized with the value 0.

- o There can only be one "first time control passes completely through a declaration."

- o It is not possible to get to the statement following the declaration without control passing completely through the declaration, so there is no possibility that the variable will be uninitialized in the following statement.

- o When entering the declaration, we won't know if this will be the first time control passes completely through, so we must compute the initializing expression each time we enter when the variable has not yet been initialized.

- o If the processor completes computing the initializing expression, and the variable has already been intialized, it must discard the computed value because only the first time through should do the initialization.

The return value from the function f the first time "control passes completely through the declaration" is 0.

This contradicts the example from the ARM (page 92)

```
int foo(int i) {
        static int s = foo(2*i);
        return i+1; // <<==
}
```

should result in an infinite loop or other undefined behavior (due to integer overflow), because there is no way to reach the marked line without s initialized, and there is no way to initialize s without reaching the marked line.

Resolution:

6.7 para 4 has been modified to say:

"If control re-enters the declaration (recursively) while the object is being initialized, the behavior is undefined. [Example:

```
int foo(int i)
{
    static int s = foo(2*i);  // recursive call - undefined
    return i+1;
```

```
             }
           --end example]
          "
  Requestor:      Neal M Gafter
  Owner:          Josee Lajoie (Initialization)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   725
  Title:          Can a local object be initialized before the first time
                  control passes through its declaration?
  Section:        6.7 [stmt.dcl]
  Status:         closed
  Description:
          6.7/4 says:
          "A local object with static storage duration not initialized
           with an integral constant-expression is initialized the first
           time control passes through its declaration..."

          This disallows early initialization of:

            struct A { int b; int c; };
            int y;
            void f() {
               static A a = { 1, 2 };
               static float x = 1.0 / 3.0;
               static int *z = &y;
            }

          Shouldn't 6.7 agree with 3.6.2 as much as possible, including
          optional early initialization?
  Resolution:
          6.7 para 4 was modified to say:
          "An implementation is permitted to perform early initialization
of
           other local objects with static storage duration under the  same
           conditions that an implementation is permitted to statically
           initialize an object with static storage duration in namespace
           scope (_basic.start.init_).  Otherwise  such an object is
           initialized the first time control passes through its
declaration;"
  Requestor:      Bill Gibbons
  Owner:          Josee Lajoie (Initialization)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   671
  Title:          Does template instantiation happen during parser
ambiguity
                  resolution?
  Section:        6.8 [stmt.ambig]
  Status:         closed
  Description:
          6.8 [stmt.ambig] para 3:
          "[Note: because the disambiguation is purely syntatic, template
           instantiation does not take place during the diambiguation
           step.]

          Is the compiler allowed or required to instantiate during
          parser ambiguity resolution?  The WP would imply "no" but how
          is one otherwise to deal with "x<y>::z" during ambiguity
          resolution?
  Resolution:
          6.8 para 3 now says:
          "Class templates are instantiated as necessary to determine if a
           qualified name is a type-name."
  Requestor:      Neal Gafter
  Owner:          Bill Gibbons / John Spicer (Templates)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  =========================================================================
====
  Chapter 7 - Declarations
  --------------------------
  Work Group:     Core
  Issue Number:   726
  Title:          inline functions must be declared inline in all
                  translation units - is a diagnostic required?
  Section:        7.1.2 [dcl.fct.spec]
  Status:         closed
  Description:
          7.1.2, para. 4:
          "If a function with external linkage is declared inline in one
           translation unit, it shall be declared inline in all translation
           units in which it appears."

          Should this be followed by 'no diagnostic required', or is this
          subsumed by the ODR requirement?
  Resolution:
          It was specified that no diagnostic is required for a violation
of
          the rule above.
  Requestor:      Roly Perera
  Owner:          Josee Lajoie (ODR)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   727
  Title:          In which namespace are names in extern block
                  declarations and function block declarations looked up?
  Section:        7.3.1.2 [namespace.memdef]
  Status:         closed
  Description:
          int f();
          int i;
          namespace N {
              int g() {
                int f();      // is this ::f or N::f?
                extern int i; // is this ::i or N::i?
              }
          }

         In which enclosing namespace scopes are names in a extern local
         declaration or a function declaration looked up?
         Shouldn't this follow what has been decided for friends?
         i.e. if the name is not found in the immediate enclosing
         namespace, the block scope declaration refers to a member of the
         immediately enclosing namespace.
  Resolution:
          3.5 para 6 now says:
```

```
          "The name of a function declared in block scope, and the name of
an
           object declared by a block scope extern declaration, have
linkage.
           If there is a visible declaration of an entity with linkage
having
           the same name and type, _ignoring entities declared outside the
           innermost enclosing namespace scope_, the block scope
declaration
           declares that same entity and receives the linkage of the
previous
           declaration."

          Only the immediately enclosing namespace is searched, just as it
is
          the case for friends.
  Requestor:      Bill Gibbons
  Owner:          Josee Lajoie (Name Lookup)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   673
  Title:          Does a using-declaration for an enum type declare aliases
for
                  the enumerator names as well?
  Section:        7.3.3 [namespace.udecl]
  Status:         closed
  Description:
          namespace N {
                  enum E { a, b };
          }
          using N::E;
          int i = a; //ok? Is the enumerator 'a' visible here?
  Resolution:
          The following note was added to 7.3.3 para 2:
          "[Note: only the specified name is so declared; specifying an
           enumeration name in a using-declaration does not declare its
           enumerators in the  using-declaration's declarative region.  ]"
  Requestor:
  Owner:          Josee Lajoie (Name Lookup)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   612
  Title:          name look up and unnamed namespace members
  Section:        7.3.4 [namespace.udir]
  Status:         closed
  Description:
          Should static not be deprecated?

          paragraph 5 says:
          "If name look up finds a declaration for a name in two different
           namespaces, and the declarations do not declare the same entity
           and do not declare functions, the use of the name is ill-
formed."

          Consider the program:

             struct S { };
             static int S;
```

```
   int foo() { return sizeof(S); }
```

The sizeof will resolve to the static int S, because nontypes are favored.

The standard says that unnamed namespaces will deprecate the use of static so we should be able to rewrite the program as:

```
   struct S { };
   namespace {
      int S;
   }
   int foo() { return sizeof(S); }
```

However, the sizeof becomes ambiguous according to 7.3.4 para 5 because the two S are from different namespaces. Is this right? Doesn't this mean that static should not be deprecated?

Resolution:

At the Hawaii meeting, the core WG decided that this situation was acceptable.

Requestor:
Owner: Josee Lajoie (Name Look up)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 728
Title: How are extern "C" objects declared or defined?
Section: 7.5 [dcl.link]
Status: closed
Description:

```
extern "C" int i;
```

Does 'extern' influences whether this is a declaration or a definition?
If it is a definition, then how does a declaration look like?
How do you declare 'i' in many translation units?

```
extern "C" extern int i; // ??
```

The WP needs to be clearer about this.

Resolution:

7.5 para 7 says:

"The form of linkage-specification that contains a braced-enclosed declaration-seq does not affect whether the contained declarations are definitions or not (_basic.def_); the form of linkage-specification directly containing a single declaration is treated as an extern specifier (_dcl.stc_) for the purpose of determining whether the contained declaration is a definition.
[Example:

```
  extern "C" int i; // declaration
  extern "C" {
         int i;    // definition
  }
```

--end example]
"

Requestor:
Owner: Josee Lajoie (extern "C")

```
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  ===========================================================================
====
   Chapter 8 - Declarators
  -------------------------
  ===========================================================================
====
   Chapter 9 - Classes
  ---------------------
  ===========================================================================
====
   Chapter 10 - Derived classes
  -----------------------------
  Work Group:     Core
  Issue Number:   674
  Title:          How do using-declarations affect class member lookup?
  Section:        10.2 [class.member.lookup]
  Status:         closed
  Description:
          10.2 para 2:
          "First, every declaration for the name in the class and in each
           of its base class sub-objects is considered. A member name f in
           one sub-object B hides a member name f in a sub-object A if A
           is a base class sub-object of B. Any declarations that are so
           hidden are eliminated from consideration. If the resulting set
           of declarations are not all from sub-objects of the same type,
           or the set has a nonstatic member and includes members from
           distinct sub-objects, there is an ambiguity and the program is
           ill-formed."

          struct A { static int i; };  // NOTE: static member
          struct B : A { };
          struct C : A { using A::i; };
          struct D : B, C { void foo(); };
          void D::foo()
          {
              i;   // ambiguous?
          }

          Is this ambiguous?
          The declarations found are from sub-objects of different types;
          however, the declarations found refer to the same static member
          from a sub-object of type A.
  Resolution:
          The following sentence was added to 10.2 para 2 to clarify what
          happens with base class members introduced with using-
declarations:
          "Each of these declarations that was introduced by a
           using-declaration is considered to be from each sub-object of C
           that is of the type containing the declaration designated by the
           using-declaration."
  Requestor:
  Owner:          Josee Lajoie (Name Lookup)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   675
  Title:          How do using-declarations influence the selection of a
final
```

virtual function overrider?
Section: 10.3 [class.virtual]
Status: closed
Description:

If a virtual function final overrider can be introduced by a using-declaration, the WP should provide an example of what happens for hierarchies with multiple inheritance. The result in some situations will be somewhat surprising for the users.

```
class A {
        void f();
};

class B {
        virtual void f() = 0;
};

class C: public A, public B {
        using A::f; // override B::f from A::f
} c;

main()
{
        c.f(); // call A::f
}
```

Resolution:

10.3 para 2 was modified to say that names introduced by using-declarations are ignored when determining the final overrider.

Requestor: Neal Gafter
Owner: Josee Lajoie (Name Lookup)
Emails:
core-7060
Papers:

. . . .

==

Chapter 11 - Member Access Control

Work Group: Core
Issue Number: 731
Title: Do functions first declared as friends still have external linkage?
Section: 11.4 [class.friend]
Status: closed
Description:

11.4p4:

> "A function first declared in a friend declaration has external linkage"

Isn't this inconsistent with the dropping of insertion? Since the declaration isn't inserted into the surrounding context, why shouldn't the linkage be left unspecified until the actual declaration that introduces the name?

Resolution:

At the Hawaii meeting, the core WG decided that the current rule was acceptable.

Requestor: Mike Miller
Owner: Steve Adamczyk (Access)
Emails:
Papers:

. . . .

```
. .
  =========================================================================
====
   Chapter 12 - Special Member functions
  ---------------------------------------
  Work Group:     Core
  Issue Number:   732
  Title:          Should "explicit" be allowed on type conversion
                  operators?
  Section:        12.3 [class.conv]
  Status:         closed
  Description:
          Steve Clamage:
          The question of whether "explicit" should be allowed on type
          conversion operators as well as on constructors has come up a
          few times in comp.std.c++.  Pablo Halpern, quoted below,
          presented what I think is a good argument in favor of allowing
          it.  I don't know of any arguments against it, other than its
          utility.  Pablo's example addresses utility.

          In article 3775050@news.ma.ultranet.com,
          phalpern@truffle.ma.ultranet.com (Pablo Halpern) writes:
          >
          > ...
          >
          >class Rational
          >{
          >public:
          >  Rational(long numerator, long denominator = 1);
          >  explicit Rational(double = 0.0);
          >
          >  // Steve Clamage suggests this:
          >  double to_double() const;  // May lose precision
          >  ...
          >};
          >
          >template <class T>
          >void process(T x)
          >{
          >  double y1 = someFunc(static_cast<double> x);  // Option 1
          >  double y2 = someFunc(x.to_double());          // Option 2
          >  // ...
          >}
          >
          >void f()
          >{
          >  Rational a(5.0);
          >  double b(5.0);
          >  long c(5);
          >
          >  process(a);  // Option 1 fails, Option 2 works
          >  process(b);  // Option 1 works, Option 2 fails
          >  process(c);  // Option 1 works, Option 2 fails
          >}
          >
          > I don't want to define an implicit conversion from Rational
          > to double and it is not reasonable to ask me to specialize f()
          > for every type that is castable to double (especially since
          > some such types may not be written yet).  So there is no way
          > to write f() such that it works for build-in types, implicitly
          > castable classes, and classes with to_double() functions.
          > Worse, if I use a 3rd-party class that supplies a conversion
          > function called asDouble() instead of to_double(), my template
          > becomes totally useless.
```

>
> Allowing explicit conversion operators provides a convention
> for naming explicit conversion functions which works for both
> built-in and user-defined types. It is also orthogonal to
> explicit constructors and makes it easier to teach C++.
>
> Principle: When considering work-arounds for lack of a
> language feature (e.g. to_double() is a work around for the
> lack of explicit operator double()), consider whether the
> work-around will work in a template class or function.

Resolution:

At the Hawaii meeting, the core WG decided that this was a request for extensions and would not be handled at this late stage.

Requestor: Steve Clamage
Owner: Steve Adamczyk (Type Conversions)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 138 (WMM.89)
Title: When are default ctor default args evaluated for array elements?
Section: 12.6 [class.init] Initialization
Status: closed

Description:

From Mike Miller's list of issues.
WMM.89. Are default constructor arguments evaluated for each element of an array or just once for the entire array?

```
        int count = 0;
        class T {
                int i;
        public:
                T ( int j = count++ ) : i ( j ) {}
                ~T () { printf ( "%d,%d\n", i, count ); }
        };
        T arrayOfTs[ 4 ];
```

Should this produce the output :-

```
        0,4
        1,4
        2,4
        3,4
```

or should it produce :-

```
        0,1
        0,1
        0,1
        0,1
```

Proposed Resolution:

8.3.6[dcl.fct.default] para 9 says:
"Default arguments are evaluated at each point of call before the entry into a function."
This should also be true if the function call is implicit.
That is, the test case above should produce the first output suggested above.

Para 9 should be clarified to say that it also applies to functions that are implicitly called.

Resolution:

Para 9 now says that the arguments are evaluated each time the function is called.

Requestor: Mike Miller / Martin O'Riordan

```
  Owner:          Josee Lajoie (Object Model)
  Emails:
          core-668
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  ==========================================================================
====
   Chapter 13 - Overloading
  -------------------------
  Work Group:     Core
  Issue Number:   662
  Title:          Do cv-qualifiers on the class object influence the
                  operator() called?
  Section:        13.3.1.1.2 [over.call.object]
  Status:         closed
  Description:
          Should this be unambiguous?

          typedef int (*pf)(char);
          int foo(char);

          struct S {
             operator pf() const { return c1; }
             operator pf() volatile { return c2; }
          };
          void f() {
              volatile S vs;
              vs('a');
          }

          If so, paragraph 2 needs to be changed to only allow
          conversion functions whose cv-qualifiers are at least as
          qualified as the expression's qualifiers.
  Resolution:
          Paragraph 2 now says:
          "where cv-qualifier is the same cv-qualification as, or a
_greater
           cv-qualification than_, cv,..."
  Requestor:
  Owner:          Steve Adamczyk (Type Conversions)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   684
  Title:          The ranking for implicit conversion sequences for pointer
                  types should take into account qualification conversions
in
                  4.4.
  Section:        13.3.3.2 [over.ics.rank]
  Status:         closed
  Description:
          Section 13.3.3.2 [over.ics.rank] says:

          Two implicit conversion sequences of the same form are
          indistinguishable conversion sequences unless one of the
following
          rules apply:

          - Standard conversion sequence S1 is a better conversion sequence
            than standard conversion sequence S2 if
```

[...]

- S1 and S2 differ only in their qualification conversion and they yield types identical except for cv-qualifiers and S2 adds all the cv-qualifiers that S1 adds (and in the same places) and S2 adds yet more cv-qualifiers than S1, or if not that,

[...]

This may predate the Koenig & Smith papers on safe cv-qualification conversions in multi-level pointer and reference types. Shouldn't the ranking be based on whether one type can safely be converted into the other? Of course that involves more than just "more qualifiers".

Resolution:

The bullet above was changed to:

"- S1 and S2 differ only in their qualification conversion and yield similar types T1 and T2 (_conv.qual_), respectively, and the cv-qualification signature of type T1 is a proper subset of the cv-qualification signature of type T2, ... "

Requestor: Bill Gibbons
Owner: Steve Adamczyk (Type Conversions)
Emails: core-6996
Papers:

. . . .

Work Group: Core
Issue Number: 685
Title: What is the ranking of a user-defined conversion that combines a pointer conversion with casting away cv-qualifiers?
Section: 13.3.3.2 [over.ics.rank]
Status: closed

Description:

5.4 para 5 says:

The conversions performed by
-- a const_cast (_expr.const.cast_),
-- a static_cast (_expr.static.cast_),
-- a static_cast followed by a const_cast,
-- a reinterpret_cast (_expr.reinterpret.cast_), or
-- a reinterpret_cast followed by a const_cast,
can be performed using the cast notation of explicit type conversion. The same semantic restrictions and behaviors apply.

This means that this code is well-formed:

```
struct A {
        operator const char *();
} a;

main () {
                // const_cast<char *>(static_cast<const char*>(a))
        char *p = (char *) a;
}
```

In which case the overloading rules in chapter 13 need to describe what happens in this case:

```
struct A {
        operator const char *();
        operator const volatile char *();
} a;

main () {
        char *p = (char *) a;
}
```

Resolution:

The following text was added at the end of 5.4 paragraph 5: "If a conversion can be interpreted in more than one way as a static_cast followed by a const_cast, the conversion is ill-formed."

Requestor: Jason Merrill
Owner: Steve Adamczyk (Type Conversions)
Emails: core-7023
Papers:

. . . .

==

Chapter 14 - Templates

Work Group: Core
Issue Number: 735
Title: Semantics for some forms of the template parameter missing?
Section: 14.1 [temp.param]
Status: closed
Description:

The syntax allows

```
   template <class>
```

The semantics should say what happens in this case.

Resolution:

It is permitted by the syntax.
The opinion of the core WG is that nothing special needs to be said for this case.

Requestor:
Owner: Bill Gibbons/John Spicer (Templates)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 738
Title: Can a template parameter be declared as a friend?
Section: 14.6.1 [temp.local]
Status: closed
Description:

14.6.1:5
"A template parameter shall not be redeclared within its scope.."
Does this ban the following friend declaration?

```
template <class T> struct B {
   friend class T;      //?

   friend void T::f();  //ok
};
```

Resolution:
7.1.5.3 p5 was changed to say:
"If the identifier resolves to a typedef-name or a template type-parameter, the elaborated type-specifier is ill-formed. [Note: this implies that, within a class template with a template type-parameter T, the declaration "friend class T;" is ill-formed.]

Requestor: Jason Merrill
Owner: Bill Gibbons/John Spicer (Templates)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 676
Title: When is a template instantiated?
Section: 14.7.1 [templ.inst]
Status: closed
Description:

14.7.1 para 3 says:
"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 template specialization is implicitly instantiated."

'In a way that involves overload resolution' is not very precise.

Consider the following case:

```
template <class T> class foo {
public:
   operator int();
};

void bar(int);
void bar(float);
void bar(foo<int>&);

void foo_bar(foo<int>& fi)
{
        bar(fi);
}
```

Is the template instantiated during overload resolution for the call to bar?

Suppose that bar(foo<int>&) isn't there, is the instantiation still required?

What about calls to friend functions:

```
    extern void foo(int&);
    template <class T> class X {
        friend void foo(X&);
    };
    void bar(X<int>& t) {
        foo(t); // is X<int> instantiated?
                // If not, does this call fail?
```

```
              }

          -----------
          The description in 14.7.1 should be improved to clarified these
          cases.
  Resolution:
          The following text was added to 14.7.1 paragraph 3:
          "If the overload resolution process can determine the correct
           function to call without instantiating a class template
definition,
           it is unspecified whether that instantiation actually takes
place.
           [Example:
                  template<class T> struct S {
                          operator int();
                  };
                  void f(int);
                  void f(S<int>&);
                  void f(S<float>); // instantiation of S<float> allowed
                                    // but not required

                  void g(S<int>& sr)
                  {
                          f(sr);   // instantiation of S<int> allowed
                                   // but not required
                  }
          --end example]
  Requestor:      Neal Gafter
  Owner:          Bill Gibbons/John Spicer (Templates)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   739
  Title:          How does argument deduction works if operator T is a
                  member template?
  Section:        14.8.2 [temp.deduct]
  Status:         closed
  Description:
          class C {
              template <class T> operator T();
          };
          How does template deduction works for T?
          Can the template argument be a base class of the class
          converted to?
          Can the template argument be a type that can be converted to
          the target type using a standard conversion?
          Or must the template argument be exactly the type to which the
          object of type C is converted?
  Resolution:
          Core 3 decided that  the current WP is clear enough.
  Requestor:
  Owner:          Bill Gibbons/John Spicer (Templates)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  ==========================================================================
====
   Chapter 15 - Exception Handling
  ---------------------------------
  Work Group:     Core
  Issue Number:   678
  Title:          Can the exception object created by a throw expression
have
                  array type?
  Section:        15.1 [except.throw]
  Status:         closed
  Description:
              try {
                  int a[5];
                  throw a;
              }
              catch (int (&array)[5]) { }

          Does the handler catch the exception?  Or is an array-to-pointer
          conversion applied to the operand of the throw expression,
meaning
          that the exception thrown has type pointer to int and that the
          handler does not catch the exception?

          15.1 para 3 refers to the subclause on function calls (5.2.2) and
to
          its description of conversions on function call arguments to
describe
          the conversions that apply to a throw expression.
          5.2.2 says that whether the array-to-pointer conversion is
applied to
          an argument in a function call depends on the type of the
function
          parameter.
          In the case of the throw expression, either the conversion is
always
          performed or it is never performed, but I don't believe saying
that
          it depends on the type of the handler makes any sense.  I think
this
          should be clearer in 15.1.
  Resolution:
          The array to pointer conversion always takes place.
          See 15.1 para 3:
          "The throw-expression initializes a temporary object, the type of
           which is determined by removing any top-level cv-qualifiers from
the
           static type of the operand of throw and adjusting the type from
           "array of T" or "function returning T" to "pointer to T" or
"pointer
           to function returning T", respectively".
  Requestor:
  Owner:          Bill Gibbons (Exceptions)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   740
  Title:          Can an exception specification appear in a reference
                  declaration?
  Section:        15.4 [except.spec]
  Status:         closed
  Description:
          15.4p1 permits exception specifications in function and pointer
          declarations but not in reference declarations.  Was this
          intentional?

          Likewise, is "pointer declaration" intended to include
          pointer-to-member declarations?
```

```
              void f() throw(int);                        // okay
              void (*fp)() throw(int) = f;                // okay
              void (&fr)() throw(int) = f;                // ill-formed --
why?

              struct A { void f() throw(int); };

              void (A::*pmf)() throw (int) = &A::f;       // is this
permited?
  Resolution:
          Yes.
          15.4 para 1: "An exception specification shall appear only in ...
          a reference ..."
  Requestor:      John Spicer
  Owner:          Bill Gibbons (Exception Handling)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   741
  Title:          The definition of uncaught_exception does not take into
                  account nested exceptions
  Section:        15.5.3 [except.uncaught]
  Status:         closed
  Description:
          15.5.3 para 1:
          "The predicate
              bool uncaught_exception();
           returns true after completing evaluation of the object to be
           thrown until completing the initialization of the
           exception-declaration in the matching handler (_lib.uncaught_).
           This includes stack unwinding (_except.ctor_)."

          Which of the following two descriptions is the correct
          interpretation of uncaught_exception() returning true?

          1. Returns true if there is *any* exception that is uncaught.
             In other words it returns true if terminate() *might* be
             called should the search for a matching handler reach an
             uncaught exception.

          2. Returns true only when immediately inside of an uncaught
             exception.  In other words, any attempt to throw an object
             will result in terminate() being called.

             Example of rule 2:
             #include <exception.h>
             #include <assert.h>
             struct A {
                A(){}
                A(const A&) {
                   // A throw here will cause terminate() to be called
                   assert(std::uncaught_exception() != false);
                   try {
                      // A throw here will not cause terminate() to be
                      // called
                      assert(std::uncaught_exception() == false);
                      throw 1;
                   }
                   catch (...){
                      // A throw here will cause terminate() to be called
                      assert(std::uncaught_exception() != false);
                   }
                }
             };

             int main()
             {
                A a;
                try {
                   throw a;
                }
                catch (...){}
             }
  Resolution:
  Requestor:      John Spicer
  Owner:          Bill Gibbons (Exception Handling)
  Emails:
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  =====================================================================
====
   Chapter 16 - Preprocessing Directives
  ----------------------------------------
  Work Group:     Core
  Issue Number:   679
  Title:          "Shall" is used incorrectly in clause 16
  Section:        clause 16
  Status:         closed
  Description:
          John Spicer pointed out the following:

          > There are numerous uses of "shall" in clause 16 (much of which
          > came directly from the C standard).  The problem is that
          > "shall" does not always mean the same thing in the two
          > documents (in only means the same thing when it appears in a
          > "constraint" in the C standard).
          >
          > It seems that someone should go though clause 16 and change
          > "shall" to the appropriate wording about undefined behavior.
          > If > this is not done, certain programs that are undefined in
          > C will become ill-formed in C++.
  Resolution:
          The changes suggested in 96-0218/N1036 were applied.
  Requestor:      John Spicer
  Owner:          Tom Plum (C Compatibility)
  Emails:
          compat-324
  Papers:
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. .
  Work Group:     Core
  Issue Number:   742
  Title:          Should __STDC__ be in the list of predefined macros?
  Section:        16.8 [cpp.predefined]
  Status:         closed
  Description:
          Section 16.8 [cpp.predefined] lists the predefined macros, and
          therefore defines what the standard means by "predefined macro".
          __STDC__ is on this list, but its definition is

            "__STDC__
             Whether __STDC__ is defined and if so, what its value is,
             are implementation-defined."
```

So it's a "predefined macro", but it might not be defined. (?!?!?). Being a "predefined macro" __STDC__ IS covered by the later constraint

"2 The values of the predefined macros (except for __LINE__ and __FILE__) remain constant throughout the translation unit.

3 None of these macro names, nor the identifier defined, shall be the subject of a #define or a #undef preprocessing directive. All prede- fined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore."

So if the implementation decides not to define __STDC__, must it "remain constant throughout the translation unit"? Does this apply if the implementation does decide to provide a definition for __STDC__? Or is that up to the implementation as well? I'd like the implementation to have these freedoms; right now it just isn't clear what was intended.

Resolution:
See 96-0218/N1036.

Requestor: Neal Gafter
Owner: Tom Plum (C Compatibility)
Emails:
Papers:

. . . .

==

Annex C - Compatibility

Work Group: Core
Issue Number: 681
Title: The type of string literals is array of const char - this has implicitions for C compatibility and should be in Annex C
Section: C.2.1 [diff.lex]
Status: closed

Description:
Jonathan Schilling wrote the following:
The WP changes for the motion at Stockholm to change the type of string literals didn't include anything for Annex C.2. Something is needed, since this represents a new incompatibility with C. If no one has written up the new entry, I propose the attached.

Proposed Resolution:
C.2.1 Clause 2: lexical conventions [diff.lex]

(insert as paragraph 4)

Subclause 2.13.4

Change: Type of string literal is changed from array of char to array of const char, and type of wide string literal from array of wchar_t to array of const wchar_t.

Rationale: This improves the consistency of the C++ type system.

Effect on original feature: Change to semantics of well-defined feature.

Difficulty of converting: Syntactic transformation. The most common cases are handled by a new but deprecated standard conversion:

```
char* p = "abc";                // valid in C, deprecated in C++
char* q = expr ? "abc" : "de";  // valid in C, invalid in C++
```

How widely used: Common.

Resolution:

This difference is now listed in C.2.1.

Requestor: Jonathan Schilling
Owner: Tom Plum (C Compatiblity)

Emails:

compat-350

Papers:

. . . .
. .