Accredited Standards Committee X3
Information Processing Systems
Operating under the procedures of
American National Standards Institute

Doc No: X3J16/95-0111 WG21/N0711
Date: May 30, 1995
Project: Programming Language C++
Ref Doc:
Reply to: Josee Lajoie
(josee@vnet.ibm.com)

Core WG List of Issues

Core1

Linkage / ODR

3.2 [basic.def.odr]:
- 113: Can inline functions refer to static global variables?
- 427: When is a diagnostic required when a member function used is not defined?
- 428: ODR for block scope extern declarations

3.5 [basic.link]:
- 461: Do namespace names have linkage?

7.1.1 [dcl.stc]:
- 437: How do extern declarations link with previous declarations?
- 473: linkage of local static variables needs to be better described

7.1.2 [dcl.fct.spec]:
- 454: Can two inline functions call one another?

7.1.3 [dcl.typedef]:
- 502: What is the linkage name of unnamed enum types introduced by a typedef?

7.5 [dcl.link]:
- 78: Linkage specification and calling protocol
- 420: Linkage of C++ entities declared within `extern "C"'
- 486: What is the effect of multiple linkage declarations?

Memory Model

3.9 [basic.types]:
- 192: Should a typedef be defined for the type with strictest alignment?
- 511: The requirements on alignment restrictions are incorrect
- 464: Can memcpy be used to copy pointer to members?
- 522: numeric_limits and fundamental types

5.3.5 [expr.delete]:
- 471: When can an implementation change the value of a delete expression?
- 93: Deleting the "current object" (this) in a member function

new / delete

3.7.3.1 [basic.stc.dynamic.allocation]:
- 463: Can operator new/delete with placement be declared in a named namespace?

5.3.4 [expr.new]:
- 435: Can reference types be newed?
- 436: What is the type of a new expression allocating an array?
- 453: Can operator new be called to allocate storage for temporaries, RTTI or exception handling?

5.3.5 [expr.delete]:
- 340: Which delete operator must be used to delete objects and arrays?
- 416: Can a delete expression be of abstract type?
- 426: Deleting arrays if dynamic type differs from type of delete expression
- 470: Deleting a pointer allocated by a new with placement

Object Model

1.6 [intro.object]:
421: What is a complete object? a sub-object?
5.2.1, 5.2.5, 5.3.1, 5.3.2, 5.7:
417: Should pointer arithmetic be allowed for pointer-to-abstract?
8.5 [dcl.init]:
476: Can objects with "indeterminate initial value" be referred to?
10.1 [class.mi]:
11: How do retrictions on member mapping apply with multiple inheritance?
442: Can a class have a direct and an indirect base of the same type?
481: Can two base class subobjects be allocated at the same address?
10.3 [class.virtual]:
447: When should a class without a final overrider be ill-formed?
11.1 [class.access.spec]:
22: Must implementations respect access restrictions?
11.2 [class.access.base]:
284: Access to base class ctor/dtor
12.1 [class.ctor]:
379: Invoking member functions which are "not inherited"

Construction / Destruction

3.6.1 [basic.start.main]:
462: Calling exit from a destructor for a global variable
484: When must objects of static storage duration be destroyed wrt to calls to functions registered with atexit?
3.6.2 [basic.start.init]:
429: Order of initialization of reference variables
3.6.3 [basic.start.term]:
430: Order of destruction of local static variables
520: Order of object destruction and atexit
521: abort and destruction of objects of automatic storage duration
6.7 [stmt.dcl]:
524: Exactly when are objects of automatic storage duration destroyed?
12.4 [class.dtor]:
485: Order of destruction of base classes and members
293: Clarify the meaning of y.~Y
508: When is a destructor implicitly-defined?
12.6 [class.init]:
138: When are default ctor default args evaluated for array elements?
12.6.2 [class.base.init]:
359: Timing of Evaluations in Base and Member Initializations

Class Copy

12.8 [class.copy]:
95: Volatility, copy constructors, and assignment operators

Temporaries

5.18 [expr.comma]:
499: When are temporaries destroyed when created by the left expression of the comma operator?
12.2 [class.temporary]:
477: When can an implementation create temporaries?
516: What is the lifetime of "helping" temporaries?
507: Must a temporary initializing an object be destroyed _as soon as_ it has been copied?
509: How does "destroy in reverse order of creation" work for temporaries

bound to references vs temporaries destroyed at the end of the initialization?

RTTI

5.2.6 [expr.dynamic.cast]:
  468: How does dynamic_cast to void* work for non-polymorphic types?
5.2.7 [expr.typeid]:
  483: What does a compiler know about type_info if the header isn't included?

Core Editorial
----

3 [basic]:
  460: Definition for the term "variable"
3.3.1 [basic.scope.local]:
  509: The controlling expression of a do statement cannot declare anything
9.7 [class.bit]:
  525: Implementation specific mapping of bitfields

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

Lexical Analysis

2.3 [lex.pptoken]:
  519: The grammar does not provide a production for pp-number
2.9.2[lex.ccon]:
  459: Octal sequences
2.9.3 [lex.fcon]:
  506: Is a program containing a non-representable floating point constant ill-formed?

Name Look Up

3.4 [class.scope]:
  510: Function overloads may come from different scopes
5.1 [expr.prim]:
  433: What is the syntax for explicit destructor calls?
5.2.4 [expr.ref]:
  452a: How does name look up work after . or -> for namespace names or template names?
5.3.4 [expr.new]:
  469: Better name lookup rules for operator new needed
7.3.2 [namespace.alias]:
  474: Lookup of namespace names in alias declarations
7.3.4 [namespace.udir]:
  475: Lookup of namespace names in using directives
10[class.derived]:
  441: In which scope is the base class clause looked up?
10.1 [class.mi]:
  446: Can explicit qualification be used for base class navigation?
11.4 [class.friend]:
  448: Can '::' be used to declare global functions as friends?
12.5 [class.free]:
  450: How is a class operator new/delete looked up?

Types / Classes / Unions

5.6 [expr.mul]:
  489: Can the operands of the multiplicative operators be of enumeration type?
5.7 [expr.add]:
  490: Can the operands of the additive operators be of enumeration type?
5.8 [expr.shift]:
  491: Can the operands of the shift operators be of enumeration type?
5.9 [expr.rel]:
  492: Can the operands of the relational operators be of enumeration type?
5.11 [expr.bit.and], 5.12 [expr.xor], 5.13 [expr.or]:
  494: Can the operands of the bitwise operators be of enumeration type?
5.16 [expr.cond]:
  495: Can the operands of the conditional operator be of enumeration type?
5.17 [expr.ass]:
  498: Can the operands of the assignment operator be of enumeration type?
7 [dcl.dcl]:
  213: Should vacuous type declarations be prohibited?
7.1.5 [dcl.type]:
  116: Is "const class X { };" legal?
7.2 [dcl.enum]:
  503: Better semantics of bitfields of enumeration type needed
9 [class]:
  514: The list of member types a POD class cannot have is not complete
9.1 [class.name]:
  252: Where can the definition of an incomplete class object appear?
9.2 [class.mem]:
  479: How can typedefs be used to declared member functions?
9.6 [class.union]:
  335: Can unions contain reference members?
  266: Access specifiers in union member list
  105: How can static members which are anon unions be initialized?
  478: Can a union constructor initialize multiple members?
  505: Must anonymous unions declared in unnamed namespaces also be static?

Declarators

6.7 [stmt.dcl]:
  500: Can one jump over the definition of a variable of enumeration type?
8.3.2 [dcl.ref]:
  456: What is the linkage of a cv-qualified reference to T?
  504: Isn't it ill-formed to directly declare a cv-qualified reference?
8.3.4 [dcl.array]:
  457: What is the linkage of a cv-qualified array?
8.3.5 [dcl.fct]:
  439: Are there any restrictions on the parameter before the ellipsis?
  482: Are cv-qualifiers allowed in a typedef for a function type?
9.7 [class.bit]:
   47: enum bitfields - can they be declared with < bits than required?
  267: What does "Nor are there any references to bitfields" mean?
  458: When is an enum bitfield signed / unsigned?

Lvalues

3.10 [basic.lval]:
  431: Are void expressions ever lvalues?

Type Conversions / Function Overload Resolution

4.9 [conv.fpint]:
  455: Should implicit floating->integral conversions be deprecated?
4.13 [conv.bool]:

523: Conversions from pointer to integral type
5.2.9 [expr.reinterpret.cast]:
486: Can a value of enumeration type be converted to pointer type?
487: Conversions to pointer to member functions need to be changed to allow covariant return types
5.9 [expr.rel]:
493: Better description of the cv-qualification for the result of a relational operator needed
513: Are pointer conversions implementation-defined or unspecified?
5.16 [expr.cond]:
472: lvalue-> rvalue transformation for the operands of the conditional operator needs better description
496: The cv-qualification of the result of the conditional operator needs better description
497: The conversion applied to operands of the conditional operator of class type needs to be better described
12.3.2 [class.conv.fct]:
347: Limitations on declarations of user-defined type-conversions
13.2.1.1.1 [over.call.func]:
451: Description of a call to a member function through a pointer to member is missing
13.6 [over.built]:
517: Candidate operator functions for built-in operators do not cover the type bool appropriately
518: Candidate operator function for built-in operator= for pointer to members is missing

Access Specification & Friends

11.3 [class.access.dcl]:
388: Access Declarations and qualified ids
11.5 [class.protected]:
449: restriction on protected member access should not apply to types, static members, and member constants
9.8 [class.nest]:
68: How do access control apply to members of nested classes in the definition of the owning class?
27: What is the access of nested class types declared multiple times

Pointer to Members

5.5 [expr.mptr.oper]:
488: Can a pointer to a mutable member be used to modify a const class object?
5.10 [expr.eq]:
481: What are the semantics for pointer to member comparison?

+---+
| Syntax |
+---+

5.1 [expr.prim]:
512: parsing destructors calls
465: grammar needed to support template function call
466: grammar needed to support ~int()
5.2 [expr.post]:
467: grammar does not support p->::id
6.8 [stmt.ambig]
132: Consistency between "::" and "Class::" in declarations
424: Must disambiguation update symbol tables?

Chapter 1 - Introduction

Work Group: Core
Issue Number: 421
Title: What is a complete object? a sub-object?
Section: 1.6 [intro.object] Object Model
Status: active
Description:

There appears to have been a substantive change in the definition of "sub-object" and "complete object" in the Working Paper.

Sub-objects used to include only objects representing base classes. A complete object used to include all objects (even members) that aren't base class objects of other objects. Now sub-objects include members, and complete objects exclude members. This introduces a number of unfortunate side-effects in the standard where the definitions are used.

3.8 [basic.life] p7:

"-- the original object was a complete object of type T and the new object is a complete object of type T (that is, they are not base class subobjects)."

5.2.6 [expr.dynamic.cast] p7:

"If T is ``pointer to cv void'', then the result is a pointer to the complete object pointed to by v. ...

If, in the complete object pointed (referred) to by v, v points (refers) to an public base class sub-object of a T object, ... Otherwise, 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."

5.2.7 [expr.typeid] p3

"If the expression is a reference to a polymorphic type, the type_info for the complete object referred to is the result. ...

... Otherwise, the result of the typeid expression is the value that represents the type of the complete object to which the pointer points. "

10 [derived] p3

"3 The order in which the base class subobjects are allocated in the complete object is unspecified."

5 A base class subobject might have a layout different from the layout of a complete object of the same type. A base class subobject might have a polymorphic behavior of a complete object of the same type."

10.1 [class.mi] p4

"For each distinct occurrence of a nonvirtual base class in the class lattice of the most derived class, the complete object shall contain a corresponding distinct base class subobject of that type. For each distinct base class that is specified virtual, the complete object shall contain a single base class subobject of that type."

12.7 [class.cdtor] p3:

"3 When a virtual function is called directly or indirectly from a constructor (including from its ctor-initializer) or from a destructor, the function called is the one defined in the constructor or destructor's own class or in one of its bases, but not a function overriding it in a class derived from the constructor or destructor's class or overriding it in one of the other base classes of the complete object."

...

5 When a dynamic_cast is used in a constructor (including in its ctor-initializer) or in a destructor, or used in a function called (directly or indirectly) from a constructor or destructor, if the

operand of the dynamic_cast refers to the object under construction
or destruction, this object is considered to be a complete object
that has the type of the constructor or destructor's class.

Resolution:

We need to introduce another term in the WP to describe objects that
are either member subobject or complete objects.
It was suggested before that the term "nonbase" object could be used.

Requestor: Neal M Gafter <gafter@mri.com>
Owner: Josee Lajoie (Object Model)
Emails: edit-195, edit-196
Papers:

==========

Chapter 2 - Lexical Conventions

Work Group: Core
Issue Number: 519
Title: The grammar does not provide a production for pp-number
Section: 2.3 [lex.pptoken] Preprocessing tokens
Status: active
Description:

The grammar does not provide a production for pp-number.

Resolution:
Requestor: Sean Corfield
Owner: Tom Plum
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 459
Title: Octal sequences
Section: 2.9.2 [lex.ccon] Character literals
Status: active
Description:

2.9.2 p4 says:
"There is no limit to the number of digits in either (octal or
hexadecimal) sequences."

In C, only 3 characters are allowed in octal sequences.
Should C++ and C differ in this case?
And if yes, appendix C should be updated.

Resolution:
Requestor:
Owner: Tom Plum
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 506
Title: Is a program containing a non-representable floating point
constant ill-formed?
Section: 2.9.3 [lex.fcon]
Status: active
Description:

2.9.1 [lex.icon] p3 says:
"A program is ill-formed if it contains an integer literal that cannot
be represented by any of the allowed types."

For consistency with 2.9.1, shouldn't a program containing a
non-representable floating point constant be ill-formed? (if the
exponent is too large, for example?)

Resolution:
Requestor: Erwin Unruh
Owner: Tom Plum
Emails:

Papers:
. . . . . .
==========
 Chapter 3 - Basic Concepts

Work Group: Core
Issue Number: 460
Title: Definition for the term "variable"
Section: 3 [basic] Basic concepts
Status: active
Description:
  Editorial Box 5:
  The definition for the term variable is needed.
Resolution:
  Proposal:
  "A variable is introduced by an object's declaration and the
   variable's name denotes the object."
Requestor:
Owner: Josee Lajoie (Core Editorial)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 113
Title: Can inline functions refer to static global variables?
Section: 3.2 [basic.def.odr] One Definition Rule
Status: active
Description:
    7.1.2 p3 says:
    "An inline member function must have exactly the same definition in
     every compilation unit in which it appears."

    Can an inline function accessed a file static variable?
    How can it do so and respect the statement describe above?
Resolution:
  [JL:] An inline member function must have the same definition in
  different compilation units. Therefore it cannot access a file
  static variable.
  However, the ODR rule needs to be better defined and the Core WG is
  working on this.
Requestor: Small issues / Core Language WG discussions
Owner: Josee Lajoie (ODR)
Emails:
Papers:
  94-0009/N0369
. . . . . .
Work Group: Core
Issue Number: 427
Title: When is a diagnostic required when a member function used is not
        defined?
Section: 3.2 [basic.def.odr] One Definition Rule
Status: active
Description:
    p2.4
    "If a non-virtual function is not defined, a diagnostic is
     required only if an attempt is actually made to call the
     function."

    Isn't it too severe to force an implementation to issue a
    diagnostic in this case? Shouldn't it be "no-diagnostic required"?
Requestor: Josee Lajoie
Owner: Josee Lajoie (ODR)
Emails:
Papers:

. . . . . .
Work Group: Core
Issue Number: 428
Title: ODR for block scope extern declarations
Section: 3.2 [basic.def.odr] One Definition Rule
Status: active
Description:
Is a diagnostic required if 2 extern block scope declarations declare the same name with different attributes, i.e.:

```
void f() {
  extern int a[5];
}
void g() {
  extern int a[7];
}
```

Requestor: Josee Lajoie
Owner: Josee Lajoie (ODR)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 509
Title: The controlling expression of a do statement cannot declare anything
Section: 3.3.1 [basic.scope.local] Local Scope
Status: active
Description:
3.3.1p5 says:
"Names declared in the outermost block of the controlled statement of a do statement shall not be redeclared in the controlling expression."

A 'do' does not have a 'condition', so the controlling expression cannot declare anything.
Resolution:
Delete paragraph 5.
Requestor: Erwin Unruh
Owner: Josee Lajoie (Core Editorial)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 510
Title: Function overloads may come from different scopes
Section: 3.4 [class.scope] Name look up
Status: active
Description:
3.4p2 says:
"Name look up may associate more than one declaration with a name if it finds the name to be a function name; in this case, all declarations shall be found in the same scope..."

This is incorrect since overloading across namespaces is allowed.
Resolution:
Replace the sentence above with:
"Name look up may associate more than one declaration with a name if it finds the name to be a function name; in this case, all declarations found will be from the same scope, either explicitly declared in that scope or visible in that scope because of using directives (7.3.4)."
Requestor: Erwin Unruh
Owner: Steve Adamczyk (Name look up)
Emails:
Papers:
. . . . . .

Work Group: Core
Issue Number: 461
Title: Do namespace names have linkage?
Section: 3.5 [basic.link] Program and linkage
Status: active
Description:
3.5 does not discuss namespace names.
Do namespace names have linkage?
Resolution:
Yes, namespaces have external linkage.
This ensures that

```
// ----- File 1 -----
struct A {
void f();
};

// ----- File 2 -----
namespace A {
  void f();
}
```

In this example, the class A and the namespace A both have external linkage. When files 1 and 2 are linked together, the name A does not refer to the same entity in both files. The program violates the ODR and may result in undefined behavior.

Proposal:
Add a bullet to paragraph 4:
-- a namespace (7.3)
Requestor: Kate Burleson
Owner: Josee Lajoie (Linkage)
Emails: core-5555
Papers:

. . . . . .

Work Group: Core
Issue Number: 462
Title: calling exit from a destructor for a global variable
Section: 3.6.1 [basic.start.main] Main function
Status: active
Description:
3.6.1 p4:
"Calling the function [exit] declared in <cstdlib> terminates the program without leaving the current block and hence without destroying any objects with automatic storage duration."

What happens if exit is called from the destructor for a global variable?
The global destructor will be called again for the same object when exit destroys the global objects.

[JL Note:]
Related question:
What if a global destructor throws an exception and the user-defined terminate routine calls exit?
Resolution:
Simon's proposed resolution:
Since static objects are destroyed in exit(), I would expect the results to be undefined, just like calling exit() in an atexit() function is.
Requestor: Simon Tooke
Owner: Josee Lajoie (Construction/Destruction)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 484
Title: When must objects of static storage duration be destroyed wrt to calls to functions registered with atexit?
Section: 3.6.1 [basic.start.main] Main function
Status: active
Description:

The WP (28 April 1995 printing), Section 18.3.2 states that static objects are destroyed in the reverse order of their construction after all functions registered via atexit() are called. Yet Section 6.7 Paragraph 5 seems to contradict this statement by requiring that the destructor of a local object with static storage duration be called either immediately before or as part of the calls of the atexit() functions. In particular, 6.7 allows destruction before the call of atexit functions yet 18.3.2 requires it follow such functions.

Jerry Schwarz also points out:

A) What is the correct thing to happen if atexit is called during construction of a static object?

B) What should happen if a static object is constructed during a call to a function that was registered via atexit?

```
void f() {
  static T t(1);
}
void g() {
  static T u(2);
}
main() {
  atexit(f);
  atexit(g);
  f();     // ensures 't' is constructed, 'u' is not yet
  exit(0); // now what?  calls 'g' and then 'f'
  // two questions arise:
  // 1) is 'u' destroyed?
  //    (it wasn't constructed prior to the atexit function
      //     calls)
  // 2) has 't' been destroyed prior to calling 'f' again?
 }
```

C) What if exit is called from the initialization of a static before main even begins to run?

Resolution:
Requestor: lijewski@roguewave.com
Owner: Josee Lajoie (Construction/Destruction)
Emails:

core-5596

Papers:

. . . . . .

Work Group: Core
Issue Number: 429
Title: Order of initialization of reference variables
Section: 3.6.2 [basic.start.init] Initialization of non-local objects
Status: active
Description:

Is the following initialization of "r" performed "statically" (that is, at link time) or "dynamicaly" (that is, at its turn among dynamically initialized nonlocal objects in the translation unit)?

```
extern int x;
int &r = x;
```

Resolution:

Requestor: Neal Gafter
Owner: Josee Lajoie (Construction/Destruction)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 430
Title: Order of destruction of local static variables
Section: 3.6.3 [basic.start.term] Termination
6.7 [stmt.dcl] Declaration statement
Status: active
Description:
3.6.3 says static objects are destroy in reverse order of intialization.
6.7 says destruction order is unspecified.
Which one is right?
Resolution:
Requestor: Mike Ball
Owner: Josee Lajoie (Construction/Destruction)
Emails: core-4989
Papers:

. . . . . .

Work Group: Core
Issue Number: 520
Title: Order of object destruction and atexit
Section: 3.6.3 [basic.start.term] Termination
Status: active
Description:
3.6.3 p1 says:
"If atexit is to be called, the implementation shall not destroy objects initialized before an atexit call until after the function specified in the atexit() call has been called."

Does this sentence refer to all objects (i.e. of automatic and static storage duration) or only to global objects (i.e. of static storage duration).
Resolution:
Change the sentence above to say:
"... the implementation shall not destroy global objects initialized before an atexit call until ..."
Requestor: Sean Corfield
Owner: Josee Lajoie (Construction/Destruction)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 521
Title: abort and destruction of objects of automatic storage duration
Section: 3.6.3 [basic.start.term] Termination
Status: active
Description:
3.6.3p3 says:
"Calling the function [abort] terminates the program without executing destructor for static objects and without calling the functions passed to atexit()."

This doesn't make it clear that destructors for local objects are not called either.
Resolution:
Change the sentence above to say:
"Calling the function [abort] terminates the program without calling the destructor for objects of automatic and static storage duration and without calling the functions passed to atexit()."
Requestor: Sean Corfield

Owner: Josee Lajoie (Construction/Destruction)
Emails:
Papers:

. . . . . .

Work Group: Core Language
Issue Number: 463
Title: Can operator new/delete with placement be declared in a named namespace?
Section: 3.7.3.1 [basic.stc.dynamic.allocation]
Status: active
Description:

3.7.3.1 p1 says:
"Allocation functions can be static class member functions or global functions."

However, the library only requires that:
::operator new(size_t)
::operator new[](size_t)
::operator delete(void*)
::operator delete[](void*)
be either declared in class scope or in global scope.

Does the sentence above apply to operator new/delete with placement as well? Or can operator new/delete with placement be declared in a namespace scope other than global scope?

Resolution:

Proposal:
Yes, operator new/delete with placement can be declared in a named namespace.

Requestor:
Owner: Josee Lajoie (new/delete)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 192
Title: Should a typedef be defined for the type with strictest alignment?
Section: 3.9 [basic.types] Types
Status: active
Description:

It would be useful if <new.h> provided a typedef for a name such as __strict_align_t , to describe a type whose alignment is the strictest required in this environment. It is otherwise hard to write a portable overloaded new operator. Faking it, by defining a union of several "typical" types, is not really portable, and its quiet mode of failure might be extremely puzzling, because the program would run just fine most of the time in most environments, except that in some unusual environment the program would occasionally produce an alignment error.

As WG14 and X3J11 have found out, some compilers add an alignment requirement for structures embedded inside structures, one which is even more restrictive than the scalar types!
There are no real-world guarantees about alignment, unless the committee imposes them.

ALTERNATIVE: The committee could prescribe specific requirements for alignment. E.g., in any conforming environment, no object may have an alignment requirement more restrictive than this specific type:
struct _strict_align_t { struct { long n; double d; }; };

92/12/07 NOTE: To allow the writing of portable allocators, it may also be necessary to define an __align_pointer(p) function, which returns the nearest pointer (address) value which is aligned on the strictest

boundary and is greater than or equal to the pointer value p .
Resolution:
Requestor: Tom Plum / Dan Saks
Owner: Josee Lajoie (Memory Model)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 511
Title: The requirements on alignment restrictions are incorrect
Section: 3.9 [basic.types] Types
Status: active
Description:
3.9p6 says:
"... an object is allocated at an address that is divisible by the alignment of its object type."

Alignment restrictions are implementation-defined. The restriction may be another then divisibility of the pointer. Beside, divisibility is not defined for pointers. The restriction should instead express 'the requirements of pointers that must point to objects of such type'.
Resolution:
Replace the sentence above with:
"... an object is allocated at an address that respects the alignment requirements for its type."
Requestor: Erwin Unruh
Owner: Josee Lajoie (Memory Model)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 464
Title: Can memcpy be used to copy pointer to members?
Section: 3.9 [basic.types] Types
Status: active
Description:
3.9 p3 says:
"For any scalar type T, if two pointers to T point to distinct T objects obj1 and obj2, if the value of obj1 is copied into obj2, using the memcpy library function, obj2 shall subsequently hold the same value as obj1."

Shouldn't this also be valid if T is a pointer to member type?

```
class A { void f(int); };
typedef (A::*PM)(int);

{
      A a;
      PM p = A::f;
      PM q;
      memcpy(&q, &p, sizeof(PM));
      (a.*q)(0);
}
```

Resolution:
Proposal:
Change the sentence above to say:
"For any scalar type or pointer to member type T, ..."
Requestor: Nathan Myers
Owner: Josee Lajoie (Memory Model)
Emails:
Papers:
. . . . . .
Work Group: Core

Issue Number: 522
Title: numeric_limits and fundamental types
Section: 3.9.1 [basic.fundamental] Fundamental types
Status: active
Description:
      3.9.1p1 says:
      "Specializations of the standard template numeric_limits (18.2) shall
       specify the largest and smallest values of each for an implementation."

      Not true for all fundamental types.
      void, bool, enumeration types are not included.
Resolution:
        Proposal:
      Replace the sentence above with:
      "Specializations of the standard template numeric_limits (18.2) shall
         specify the largest and smallest values of each funcdamental types
       (except for void, bool and enumeration types) for an implementation.
Requestor: Sean Corfield
Owner: Josee Lajoie (Memory Model)
Emails:
Papers:
. . . . . .
Work Group: Core Language
Issue Number: 431
Title: Are void expressions ever lvalues?
Section: 3.10 [basic.lval] Lvalues and rvalues
Status: active
Description:
        In ISO C, expressions having type void are never considered to be
        lvalues. Thus, the following code violates a constraint of the C
        standard, and conforming implementations are required to issue a
        diagnostic:

           void *vp;

           void foo () {
              &*vp;   /* constraint violation */
           }

        Are expression having THE void type and/or expressions having some
        qualified void type every considered to be lvalues in C++?
Resolution:
Requestor: Ron Guilmette
Owner: Steve Adamczyk (Lvalues)
Emails:
Papers:
. . . . . .
==
 Chapter 4 - Standard Conversions

Work Group: Core
Issue Number: 455
Title: Should implicit floating->integral conversions be deprecated?
Section: 4.9 [conv.fpint] Floating-integral conversions
Status: active
Description:
Requestor: Bjarne Stroustrup
Owner: Steve Adamczyk (Type Conversions)
Emails:
      core-4757
Papers:
. . . . . .
Work Group: Core
Issue Number: 523

Title: Conversions from pointer to integral type
Section: 4.13 [conv.bool] Boolean conversions
Status: active
Description:
      It is not clear, reading the rules in 4.5 [conv.prom] and 4.13
      that the following conversion is disallowed:
            int i;
            char *cp;
            i = cp;   // converts char* -> bool -> int ???
Requestor: Sean Corfield
Owner: Steve Adamczyk (Type Conversions)
Emails:
Papers:
. . . . . .
==========

Chapter 5 - Expressions

Work Group: Core
Issue Number: 512
Title: parsing destructors calls
Section: 5.1 [expr.prim] Primary expressions
Status: active
Description:
      5.1p7 says:
      "A class-name prefix by ~ denotes a destructor."

      There is a syntactic ambiguity on the usage of a destructor.
      The code '~X();' in the scope of a member function of class X can be
      interpreted as an explicit destructor call using the implicit this
        pointer. The other interpretation is the unary operator ~ applied
        to a function like cast.
Resolution:
Requestor: Erwin Unruh
Owner: Anthony Scian (Syntax)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 433
Title: What is the syntax for explicit destructor calls?
Section: 5.1 [expr.prim] Primary expressions
                12.4 [class.dtor] Destructors
Status: active
Description:
        Question 1:
        p10 says:
        The notation for explicit call of a destructor may be used for any
        simple type name. For example:
           int* p;
           p->int::~int();

        Must the destructor name be a qualified-id or can it be written as:
           p->~int();
        ?

        Question 2:
        Can a typedef name be used following the ~, and if so, what are the
        lookup rules?

        struct A {
                ~A(){}
        };

        typedef class A B;

```
        int main()
        {
                A* ap;
                ap->A::~A();    // OK
                ap->B::~B();    // cfront/Borland OK, IBM/Microsoft/EDG error
                ap->A::~B();    // cfront OK, Borland/IBM/Microsoft/EDG error
                ap->~B();       // OK?
        }

        This issue concerns the lookup of explicit destructor calls for
        nonclass types as well.

        typedef int I;
        typedef int I2;
        int*    i;
        i->int::~int();
        i->I::~I();
        i->int::~I();
        i->I::~int();
        i->I::~I2();

        Which of these are well formed?
Resolution:
Requestor:      John H. Spicer
Owner:          Steve Adamczyk (Name Lookup)
Emails:
Papers:
. . . . . .
Work Group:     Core Language
Issue Number:     465
Title:          grammar needed to support template function call
Section:    5.1 [expr.prim] Primary expression
Status:         active
Description:
      "id-expression" does not allow the syntax
        f<arg>
      needed for a call to a template function using explicit arguments.
Resolution:
      Possible solution:
        Add template-function-id (i.e. production for f<>) to the list of
          unqualified-ids:

            unqualified-id:
                  ...
                  template-function-id
Requestor:
Owner:          Anthony Scian (Syntax)
Emails:
Papers:
. . . . . .
Work Group:     Core Language
Issue Number:     466
Title:          grammar needed to support ~int()
Section:        5.1 [expr.prim] Primary expression
Status:         active
Description:
      The grammar does not allow for explicit destructor calls for built-in
        types:
        int* pi;
        pi->~int();
Resolution:
      Possible solution:
        unqualified-id:
```

```
            ...
          ~enum-name
          ~typedef-name
          ~simple-type-specifier
Requestor:
Owner:          Anthony Scian (Syntax)
Emails:
Papers:
. . . . . .
Work Group:     Core Language
Issue Number:   467
Title:          grammar does not support p->::id
Section:        5.2 [expr.post] Postfix expressions
Status:         active
Description:
Resolution:
      Possible solution:

        Change

      postfix-expression:
          postfix-expression . template(opt) id-expression
          postfix-expression -> template(opt) id-expression

      to

      postfix-expression:
          postfix-expression . ::(opt) template(opt) id-expression
          postfix-expression -> ::(opt) template(opt) id-expression
Requestor:
Owner:          Anthony Scian (Syntax)
Emails:
Papers:
. . . . . .
Work Group:     Core Language
Issue Number:   417
Title:          Should pointer arithmetic be allowed for pointer-to-abstract?
Section:        5.2.1, 5.2.5, 5.3.1, 5.3.2, 5.7
Status:         active
Description:
        Should pointer arithmetic and/or the sizeof operator be allowed for
        operands whose type is some pointer-to-abstract type?

        Note that if it is the committee's judgement to effect such
        restriction, these restrictions would have to be reflected in working
        paper sections 5.2.1, 5.2.5, 5.3.1, 5.3.2, and 5.7.
Resolution:
Requestor:      Ron Guilmette
Owner:          Josee Lajoie (Object Model)
Emails:
. . . . . .
Work Group:     Core
Issue Number:   452a
Title:          How does name look up work after . or -> for namespace names or
                template names?
Section:        5.2.4 [expr.ref] Class member access
Status:         active
Description:
        5.2.4 says p3:
        "If the nested-name-specifier of the qualified-id specifies a namespace
         name, the name is looked in the context in which the entire
         postfix-expression occurs."

        This is backward. One doesn't know if the name is a namespace name
```

until the name has been looked up. In which scope must the name following the . or -> operator be first looked up?

```
namespace N { }
struct S {
  class N { };
};
S s;

... s.N::b ...
```

The scope of the object-expression 's' or the scope in which the entire expression takes place?

Neal Gafter also asks:
"In the syntax

```
   p->template T<args>::x
```

in which scope(s) is T looked up?"

```
template <class X> class T { static X x; };

class C {
   template <class X> class T { static X x; };
};

C* p;
p->template T<args>::x
```

Resolution:
Requestor:
Owner: Steve Adamczyk (Name Look Up)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 468
Title: How does dynamic_cast to void* work for non-polymorphic types?
Section: 5.2.6 [expr.dynamic.cast]
Status: active
Description:

5.2.6 p7 says:
"If T is 'pointer to cv void', then the result is a pointer to the complete object pointed (referred) to by v. Otherwise the run-time check is applied ..."

Does this apply to pointers to non-polymorphic types?

```
class A { };
  class B { };
class C : public A, public B { };

  C c;
B* pb = &c;

dynamic_cast<void*>(pb);  // will this return a pointer to the object c?
```

Resolution:
Requestor:
Owner:
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 483
Title: What does a compiler know about type_info if the header isn't included?
Section: 5.2.7 [expr.typeid] Type identification
Status: active
Description:
If you don't include <typeinfo> and you do a typeid(...), do you get an error ("<typeinfo> must be included before using typeid")?
Or is the type type_info built-in in some way?
If so, why do we define it in a header?
Or perhaps the compiler predeclares type_info, but you have to include the header to get the definition, which means you could do a typeid(...) but then you couldn't do much with the result because type_info is incompletely-defined.
Resolution:
Requestor: Steve Adamczyk
Owner:
Emails:
core-5463
Papers:

. . . . . .

Work Group: Core
Issue Number: 486
Title: Can a value of enumeration type be converted to pointer type?
Section: 5.2.9 [expr.reinterpret.cast]
Status: active
Description:
5.2.9 p5 says:
"A value of integral type can be explicitly converted to pointer type."
Can a value of enumeration type be converted to pointer type?
Resolution:
Proposal:
Add to the sentence above: "... of integral type or enumeration type..."
Requestor: Bill Gibbons
Owner: Steve Adamczyk (Type Conversions)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 487
Title: Conversions to pointer to member functions need to be changed to allow covariant return types
Section: 5.2.9 [expr.reinterpret.cast]
Status: active
Description:
5.2.9 p11 says:
"Calling a member function through a pointer to member that represents a function type that differs from the function type specified on the member function declaration results in undefined behavior."

This needs to be clarified wrt pointer to member functions that differ because they represent virtual functions that differ only because of their covariant return type.
Resolution:
Proposal:
Add at the end of paragraph 2:
"... (except that an overriding virtual function with a different return type may be called (10.4))."
Description:
Erwin Unruh also notes:
There are implicit conversions, which alter the type of a 'pointer to member function' according to the class of which it is a member.
How does it interact with this sentence?

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

. . . . . .

Work Group: Core
Issue Number: 435
Title: Can reference types be newed?
Section: 5.3.4 [expr.new] New
Status: active
Description:

p2 says:
"The new-type in a new-expression is the longest possible sequence of new-declarators. This prevents ambiguities between declarator operators &, *, [], and their expression counterparts."

This indicates that:
'new int & i'
will be parsed as: (new int&)i
what does it mean to new a reference type?

Resolution:

Proposal:
No, reference types cannot be newed.
(The syntax does not allow for it).
The second sentence of 5.3.4 p2 should be changed as follows:
"This prevents ambiguities between declarator operators *, [], and their expression counterparts."

Requestor: ?
Owner: Josee Lajoie (New)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 436
Title: What is the type of a new expression allocating an array?
Section: 5.3.4 [expr.new] New
Status: active
Description:

5.3.4p6 says:
"When the allocated object is an array ... the new-expression yields a pointer to the initial element of the array."
Bill Gibbons says:
The type of the expression 'new T' should be T* (always). The decay to pointer to first element should only occur when the explicit array bound is used in the new expression. This allows template argument to be used to allocate arrays.

Resolution:
Requestor: Bill Gibbons
Owner: Josee Lajoie (New)
Emails:

core-3709

Papers:

. . . . . .

Work Group: Core
Issue Number: 453
Title: Can operator new be called to allocate storage for temporaries, RTTI or exception handling?
Section: 5.3.4 [expr.new] New
Status: active
Description:

Is it permitted for an implementation to create temporaries on the heap rather than on the stack? If so, does that require that operator new() be accessible in the context in which such a temporary is created?

Is an implementation allowed to call a replaced operator new whenever it likes (storage for RTTI, exception handling, initializing static in a library)?

Resolution:
Requestor: Mike Miller
Owner: Josee Lajoie (New)
Emails:
core-5068
Papers:

. . . . . .

Work Group: Core
Issue Number: 469
Title: Better name lookup rules for operator new needed
Section: 5.3.4 [expr.new] New
Status: active
Description:

5.3.4 is not clear regarding how the operator new is selected (name lookup) in the following situations:

o the syntax ::new and ::delete have special meaning.
new and delete are keywords and not function names when used as follows

```
int* pi = ::new int;
::delete pi;
```

we need to be explicit in 5.3.4 and 5.3.5 to say that the syntax ::new calls global operator delete and ::delete calls global operator delete.

o if a new with placement operator can be defined in a namespace scope (other than global scope), we need to describe how a new operator is looked up for a new expression that uses placement syntax.

Resolution:
Requestor: Steve Clamage
Owner: Steve Adamczyk (Name Lookup)
Emails:
Papers:

. . . . . .

Work Group: Core Language
Issue Number: 340
Title: Which delete operator must be used to delete objects and arrays?
Section: 5.3.5 [expr.delete] Delete
Status: active
Description:

Section 5.3.5 says:

> "In the ... alternative (delete object), the value of the operand of delete shall be a pointer to a non-array object created by a new expression."

This statement fail to take account of the fact that an "array object" may itself be considered to be an "individual object".

Here is an example which illustrates this point:

```
typedef int array_type[20];
void foobar ()
  {
   array_type *p = (array_type *) new array_type;
     delete p;       // undefined behavior???
  }
```

In this example, the current rules fail to make it completely clear whether or not the delete statement shown produces undefined behavior. Only one "object" is being deleted here, but that object happens to have an array type (a fact which the implementation could easily deduce from the static type of the pointer expression `p' given in the delete statement).

Bill Gibbons also notes:
This is also a problem when delete is used in template definitions.

```
         template <class T> void f(T* pt) {
              delete pt;  // What form of delete should be used here?
           }
        If T happens to be an array type, than the form of delete selected here
        is not appropriate.
Resolution:
      Bill Gibbons indicated the following:
      It should be made clear that the *syntax* of the delete expression
        must match the type of the object allocated by new *not* the syntax of
        the new expression.
Requestor:      Ron Guilmette
Owner:          Josee Lajoie (Delete)
Emails:
      core-3709
Papers:
. . . . . .
Work Group:     Core Language
Issue Number:   416
Title:          Can a delete expression be of abstract type?
Section:        5.3.5 [expr.delete] Delete
Status:         active
Description:
      Ron would like to see the following rules apply to the cast-expression
        of a delete expression:
                The referent type of the cast-expression may be an
                abstract class type, provided that this type was earlier defined
                to contain a virtual destructor.
Resolution:
Requestor:      Ron Guilmette
Owner:          Josee Lajoie (delete)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number:   426
Title:          Deleting arrays if dynamic type differs from type of delete
                expression
Section:        5.3.5 [expr.delete] Delete
Status:         active
Description:
      5.3.5p3
      "In the first alternative (delete object), if the static type of the
       operand is different from its dynamic type, the static type shall have
       a virtual destructor or the behavior is undefined. In the second
       alternative (delete array), if the dynamic type of the object to be
       deleted differs from its static type, the behavior is undefined."

      1) this needs to be more specific and say that there must exist an
           inheritance relationship between the static type and the dynamic
         type, i.e.
         "if the static type of the operand is different from its dynamic
          type, the static type shall be a base class of the operand's dynamic
          type and the static type must have a virtual destructor..."
        2) Jerry Schwarz asks for the following:
          Current WP disallows deleting an array using a pointer with a
          static type different from the type used in allocation.

            That is it prohibits

                class B {
                        virtual ~B() ;
                } ;
                class D : public B { } ;
```

```
            B* pb = new D[10] ;
            delete[] pb ;

          This used to make some sense because the overheads of figuring out
        exactly what to do might be significant.  (E.g. in the presence of
          MI pb might not even point to the start of the array.)

          But now that we have RTTI, there has to be enough information (in
          the vtbl) to determine the beginning of the array and its type.

          So I believe that this prohibition could reasonably be lifted.
Resolution:
Requestor:
Owner:          Josee Lajoie (delete)
Emails:
Papers:
. . . . . .
Work Group:     Core
Issue Number:   470
Title:          deleting a pointer allocated by a new with placement
Section:        5.3.5 [expr.delete] Delete
Status:         active
Description:
      5.3.5 p2 says:
      "... 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 without a new-placement specification, ..."

      In some situations, it is well-defined what happens even when new
      with placement was called. Do we want to prohibit these cases?

      Erwin Unruh also notes:
      The deletion of a pointer gained by a placement new must be allowed.
      Using the default operator delete for a pointer gained by the library
      placement new is undefined. However, a user may write placement news
      that allocate storage in which case using delete on a pointer returned
      by such a placement new should be well-defined.
Resolution:
Requestor:  Jason Merrill
Owner:          Josee Lajoie (delete)
Emails:
      core-5569
Papers:
. . . . . .
Work Group:     Core
Issue Number:   471
Title:          When can an implementation change the value of a delete
            expression?
Section:        5.3.5 [expr.delete] Delete
Status:         active
Description:
      5.3.5p4
        "If the expression denoting the object in a delete-expression is a
         modifiable lvalue, any attempt to access its value after the deletion
         is undefined."

      When can an implementation change the pointer value of a delete
      expression?

      inline void* operator new(void* p) { return p; }
      struct Base { virtual ~Base() {} };
      struct Immortal : Base {
            operator delete(void* p) { new(p) Immortal; }
      };
```

```
      main()
      {
            Base* bp = new Immortal;
            delete bp;
            delete bp;
            delete bp;
      }

      Is the above well-formed?
Resolution:
Requestor:  Nathan Myers
Owner:          Josee Lajoie (Memory Model)
Emails:
        core-5565
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core Language
Issue Number:   93
Title:          Deleting the "current object" (this) in a member function
Section:        5.3.5 [expr.delete] Delete
Status:         active
Description:
        In a standard conforming program, may delete be used within a
        non-static member function (or within a function which is called
        directly or indirectly by such a function) to delete the object for
        which the non-static member function was called?

        Example:

        struct S { void member (); };
        void delete_S (S *arg) { delete arg; }
        void S::member ()
        {
                delete_S (this);
        }

        If this is prohibited in a standard conforming program is a standard
        conforming implementation required to issue a diagnostic for such code?
Resolution:
      Proposal:
        The WP should say somewhere that accessing memory after is has been
      deallocated results in undefined behavior.
Requestor:      Ron Guilmette
Owner:          Josee Lajoie (Memory Model)
Emails:
        core-1650
Papers:
        94-0185/N0572
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   488
Title:          Can a pointer to a mutable member be used to modify a const
                class object?
Section:        5.5 [expr.mptr.oper]
Status:         active
Description:
      5.5 p4 says:
      "The restrictions on cv-qualification, and the manner in which
       cv-qualifiers of the operands are combined to produce the cv-qualifiers
         of the result, are the same as the rules for E1.E2..."

        It should be noted that a pointer to member that refers to a mutable
      member cannot be used to modify a const class object.
```

```
        struct S {
          mutable int i;
        };
        const S cs;
        int S::* pm = &S::i;
        cs.*pm = 88;
Resolution:
        Proposal:
      Add a note at the end of p4:
      "Note: a pointer to member that refers to a mutable member cannot be
         used to modify a member of an object of const class type."
Requestor:      Bill Gibbons
Owner:          (pointer to member)
Emails:
Papers:
. . . . . .
Work Group:     Core
Issue Number:   489
Title:              Can the operands of the multiplicative operators be of
                enumeration type?
Section:        5.6 [expr.mul] Multiplicative operators
Status:         active
Description:
        5.6 p2 says:
      "The operands of * and / shall have arithmetic type; the operands of %
         shall have integral type."
      Can the operand of the multiplicative operators be of enumeration type?
Resolution:
        Proposal:
      Change the sentence above as follows:
      "The operands of * and / shall have arithmetic or enumeration type; the
       operands of % shall have integral or enumeration type."
Requestor:      Bill Gibbons
Owner:              Steve Adamczyk (Types)
Emails:
Papers:
. . . . . .
Work Group:     Core
Issue Number:   490
Title:          Can the operands of the additive operators be of enumeration
                type?
Section:        5.7 [expr.add] Additive Operators
Status:         active
Description:
      5.7 omits to include in its description enumeration type.
Resolution:
        Proposal:
        Change the text in 5.7 as follows:
        "1 ...
         For addition, either or both operands shall have arithmetic _or
         enumeration type_, or one shall be a pointer to a completely defined
           object type and the other shall have integral _or enumeration type_.

         2 For substraction, one of the following shall hold:
         -- both operands have arithmetic _or enumeration type_;
         -- both operands are pointers to cv-qualified or cv-unqualified
            versions of the same completely defined object type; or
         -- the left operand is a pointer to a completely defined object type
            and the right operand has integral _or enumeration type_.

         3 If both operand have arithmetic _or enumeration type_, the usual
           arithmetic conversions are performed on them. ...
         ...
```

5 When an expression that has integral _or enumeration type_ is
added to, or substracted from a pointer, ...
...
Footnote 47:
In this scheme the integral _or enumeration expression_ added or
substracted from the converted pointer..."
Requestor: Bill Gibbons
Owner: Steve Adamczyk (Types)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 491
Title: Can the operands of the shift operators be of enumeration type?
Section: 5.8 [expr.shift] Shift Operators
Status: active
Description:
5.8 p1 says:
"The operands shall be of integral type and integral promotions are
performed."
Can the operand of the shift operators be of enumeration type?
Resolution:
Proposal:
Change the sentence above as follows:
"The operands shall be of integral or enumeration type and integral
promotions are performed."
Requestor: Bill Gibbons
Owner: Steve Adamczyk (Types)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 492
Title: Can the operands of the relational operators be of enumeration
type?
Section: 5.9 [expr.rel] Relational Operators
Status: active
Description:
5.9 says:
"1 ...
The operands shall have arithmetic or pointer type. ...

2 The usual arithmetic conversions are performed on arithmetic
operands. ..."

Can the operands of the relational operators be of enumeration type?
Resolution:
Proposal:
Change the sentences above as follows:
"1 ...
The operands shall have arithmetic, enumeration or pointer type. ...

2 The usual arithmetic conversions are performed on operands of
arithmetic or enumeration type. ..."
Requestor: Bill Gibbons
Owner: Steve Adamczyk (Types)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 493
Title: Better description of the cv-qualification of the result of a
relational operator needed
Section: 5.9 [expr.rel] Relational Operators

Status: active
Description:
5.9p2 says:
"Pointer conversions are performed on the 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."

This seems to imply that the result has exactly the type of one of the operands, or an unqualified version of that type. In fact, the common type may have more qualifiers than either operand type.
Resolution:
Proposal:
Change the text above to say:
"... which shall be the type of one of the operands, cv-qualified with the combined cv-qualifiers of the operands' types."
Requestor: Bill Gibbons
Owner: Steve Adamczyk (Type Conversions)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 513
Title: Are pointer conversions implementation-defined or unspecified?
Section: 5.9 [expr.rel] Relational Operators
Status: active
Description:
5.9p2 last '--' says:
"Other pointer comparisons are implementation-defined."

Comparison of unrelated pointers should be unspecified or undefined. At present it reads implementation defined, but I doubt that the exact rules can be described by a compiler vendor.
Resolution:
Proposal:
Change the text above to say:
"Other pointer comparisons are unspecified."
Requestor: Erwin Unruh
Owner: Steve Adamczyk (Type Conversions)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 481
Title: What are the semantics for pointer to member comparison?
Section: 5.10 [expr.eq] Equality operators
Status: active
Description:

```
struct B {
        int x;
        int f() { return x; }
        B(int a) :x(a) {}
  };
  struct L : B{ L(int a) : B(a) {} };
  struct R : B{ R(int a) : B(a) {} };
  struct D : L,R{ D() : L(1), R(2) {} };
  int (B::*pb)() = &B::f;
  int (L::*pl)() = pb;
  int (R::*pr)() = pb;
  int (D::*pdl)() = pl;
  int (D::*pdr)() = pr;
  assert(pdl != pdr);       // Is this true?

  D d;
  cout << (d.*pdl)();
```

```
        cout << (d.*pdr)();

      What does it mean to compare two member pointers for equality?
      Do they compare equal if they are the same member?
        Or do they compare equal if the have the same behavior?
Resolution:
Requestor:  John Max Skaller
Owner:          Josee Lajoie (Pointer to Members)
Emails:
      core-5391
Papers:
. . . . . .
Work Group:     Core
Issue Number:   494
Title:          Can the operands of the bitwise operators be of enumeration
                type?
Section:        5.11 [expr.bit.and], 5.12 [expr.xor], 5.13 [expr.or]
Status:         active
Description:
        All these subclauses say:
        "The operator applies only to integral operands."
         Can the operands of the bitwise operators be of enumeration type?
Resolution:
        Proposal:
        Change the sentence above to say:
        "The operator applies only to operands of integral or enumeration
         types."
Requestor:      Bill Gibbons
Owner:          Steve Adamczyk (Types)
Emails:
Papers:
. . . . . .
Work Group:     Core
Issue Number:   495
Title:          Can the operands of the conditional operator be of enumeration
                type?
Section:        5.16 [expr.cond] Conditional operator
Status:         active
Description:
        5.16 p3 says:
        "If both the second and the third expressions are of arithmetic type,
         then if they are of the same type the result is of that type; ..."
        Can the operands of the conditional operator be of enumeration type?
Resolution:
        Proposal:
        Change the sentence above to say:
        "If both the second and the third expressions are of arithmetic or
         enumeration type, ..."
Requestor:      Bill Gibbons
Owner:          Steve Adamczyk (Types)
Emails:
Papers:
. . . . . .
Work Group:     Core
Issue Number:   472
Title:          lvalue-> rvalue transformation for the operands of the
                conditional operator needs better description
Section:        5.16 [expr.cond] Conditional operator
Status:         active
Description:
      It is not clear when the lvalue-> rvalue transformation takes place on
      the operands of the conditional operator.

      5.16 p3 says:
```

"If both the second and the third expressions are of arithmetic type,
then if they are of the same type, the result is that type,
otherwise the usual arithmetic conversions are performed to bring them
to a common type."
Does "type" in this sentence include cv-qualifiers?

Other sentences in this paragraph seem to indicate that the
cv-qualifiers remain a part of the operands type:
"if both the second and the third expressions have type 'cv void',
the common type is 'cv void'".

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

. . . . . .

Work Group: Core
Issue Number: 496
Title: The cv-qualification of the result of the conditional operator needs better description
Section: 5.16 [expr.cond] Conditional operator
Status: active
Description:

5.16p3 says:
"...pointer conversions are performed on the pointer operands to bring
them to a common type, which shall be a cv-qualified or cv-unqualified
version of the type of either the second or the third expression.
...
if both the second and the third expressions are lvalues of related
class types, they are converted to a common type (which shall be
a cv-qualified or cv-unqualified version of the type of either the
second or the third expression)..."

This seems to imply that the result has either exactly the type of the
second or third expression, or the unqualified version of that type.
In fact, the common type may have more qualifiers than either operand
type.

Resolution:

Proposal:
Change the text above to say:
"... which shall be the type of either the second or the third
expression, cv-qualified with the combined cv-qualifiers of the
types of the second and third expression."

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

. . . . . .

Work Group: Core
Issue Number: 497
Title: The conversion applied to operands of the conditional operator of class type needs to be better described
Section: 5.16 [expr.cond] Conditional operator
Status: active
Description:

5.16p3 says:
"... if the second and the third expressions are lvalues of related
class types..."
The term "related class type" is not defined.

Resolution:

The sentence above should be changed to say:
"... are lvalues of class type, and one class is an unambiguous public
base of the other, ..."

Requestor: Bill Gibbons

Owner: Steve Adamczyk (Type Conversions)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 498
Title: Can the operands of the assignment operator be of enumeration type?
Section: 5.17 [expr.ass] Assignment operators
Status: active
Description:
5.17 p7 says:
"In += and -=, E1 shall either have arithmetic type or be a pointer to a possibly cv-qualified completely defined object type."
Can the operands of the assignment operator be of enumeration type?
Resolution:
Change the sentence above as follows:
"In += and -=, E1 shall either have arithmetic or enumeration type or be a pointer to a possibly cv-qualified completely defined object type."
Requestor: Bill Gibbons
Owner: Steve Adamczyk (Types)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 499
Title: When are temporaries destroyed when created by the left expression of the comma operator?
Section: 5.18 [expr.comma] Comma operator
Status: active
Description:
5.18 p1 says:
"All side effects of the left expression are performed before the evaluation of the right expression."
When are temporaries destroyed when created by the left expression of the comma operator?
Resolution:
Change the sentence above as follows:
"All side effects of the left expression, except for destruction of temporaries (12.2), are performed before the evaluation of the right expression."
Requestor: Bill Gibbons
Owner: (Lifetime of temporaries)
Emails:
Papers:
. . . . . .
==========
Chapter 6 - Statements

Work Group: Core
Issue Number: 500
Title: Can one jump over the definition of a variable of enumeration type?
Section: 6.7 [stmt.dcl] Declaration statement
Status: active
Description:
6.7p3 says:
"A program that jumps from a point where a local variable with automatic storage duration is not in scope to a point where it is in scope is ill-formed unless the variable has pointer or arithmetic type or is an aggregate, and is declared without an initializer."

Can one jump over a declaration for a variable of enumeration type?

Also, since an aggregate can have members with non-trivial constructors, using aggregate here is not appropriate.

Resolution:

Change the sentence above to say:
"... the variable has pointer, arithmetic, enumeration, or POD class type, and is declared without an initializer."

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

. . . . . .

Work Group: Core
Issue Number: 524
Title: Exactly when are objects of automatic storage duration destroyed?
Section: 6.7 [stmt.dcl] Declaration statement
Status: active
Description:

6.7p5 says:
"The destructor is called either immediately before or as part of the calls of the atexit functions."

What if the atexit function is not called?

Resolution:
Requestor: Sean Corfield
Owner: Josee Lajoie (Destruction)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 132 (WMM.83)
Title: Consistency between "::" and "Class::" in declarations
Section: 6.8 [stmt.ambig] Ambiguity resolution
Status: active
Description:

WMM.83. Is a change necessary for syntactic consistency between the treatment of "::" and "class::" in declarations?

```
float a;
float b;
main(){
      int (a)  ; // valid block scope redeclaration of a
      int (::b); // valid function like cast of b
}
```

Note that the reason for the "function like cast" interpretation is that "::b" can *only* be used as a reference, and never used as a declarator.

```
struct T { static a;};
int (T::a); // valid declaration and definition of T::a
main(){
      int (T::a);  // semantic error: attempt to redeclare T::a
      (int)(T::a); // cast of T::a
}
```

Since the syntax allows "T::a" to be used as a declarator, the statement: int (T::a); is interpreted as a declaration even though this declaration is not valid at block scope.
And eventhough the statement: int (T::a); is an invalid block scope declaration, it is not interpreted as an expression because it is validated as a declaration by the grammar.

Should the syntax "Class::" always be interpreted as a reference instead of a part of a declaration when placed inside block scope?

Resolution:
Requestor: Mike Miller / Jim Roskin
Owner: Anthony Scian (Syntax)
Emails:
      core-629
Papers:
. . . . . .
Work Group: Core
Issue Number: 424
Title: Must disambiguation update symbol tables?
Section: 6.8 [stmt.ambig] Ambiguity resolution
Status: active
Description:
      The question is about the following sentence from 6.8p3 [stmt.ambig]

      WP> The disambiguation is purely syntactic; that is, the meaning of the
      WP> names, beyond whether they are type-ids or not, is not used in the
      WP> disambiguation.

      On the one hand, this would imply that a trial parser needn't update a
      symbol table, since that would be processing that is not purely
      syntactic.

      On the other hand, some input would be disambiguated differently if the
      symbol table were updated during trial parsing. Symbol table updates
      would determine which names will be type-ids during the actual parse.

      To be more concrete and specific about the problem, consider the
      statement in main() in the enclosed test case. Should this be
      disambiguated as a declaration with a syntax error, or should it be
      disambiguated as a well-formed expression?

```
      struct T1
      {
              T1 operator()(int x) { return T1(x); };
              int operator=(int x) { return x; };
                T1(int) {};
      };
      struct T2
      {
              T2(int) {};
      };
      int a, (*(*b)(T2))(int), c, d;
      void main ()
      {
            // Is the following a declaration with a syntax error?
                // Or is it a semantically valid expression?
                T1(a) = 3,
              T2(4),
              (*(*b)(T2(c)))(int(d));
      }
```

Resolution:
Requestor: Neal M Gafter <gafter@mri.com>
Owner: Anthony Scian (Syntax)
Emails:
Papers:
==========
Chapter 7 - Declarations

Work Group: Core
Issue Number: 213
Title: Should vacuous type declarations be prohibited?
Section: 7 [dcl.dcl] Declarations
Status: active

Description:
      7p3: Should the draft prohibit vacuous decls like this?
                enum { };
                class { int i; };
                class { };
                typedef enum {};
Resolution:
Requestor: Tom Plum / Dan Saks
Owner: Steve Adamczyk (Types)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 437
Title: How do extern declarations link with previous declarations?
Section: 7.1.1 [dcl.stc] Storage class specifiers
Status: active
Description:
      3.5 paragraph 6:
      "The name of a function declared in a block scope or a variable declared
       extern in a block scope has linkage, either internal or external to
       match the linkage of prior visible declarations of the same name in
       the same translation unit, but if there is no prior visible declaration
       it has external linkage."

      7.1.1 paragraph 5:
      "...An object or function introduced by a declaration with an extern
       specifier has external linkage unless the declaration matches a visible
       prior declaration at namespace scope of the same object or function,
         in which case the object or function has the linkage specified by the
       prior declaration."

      These two sentences are similar but not identical. It would be better
      to eliminate gratuitous differences, or consolidate them in one place if
      possible. This is true of the whole discussion of linkage in these
      two sections.

      Consider the following translation unit:

      static x; /* internal linkage */

      void f() {
        auto x; /* no linkage */
        {
          extern x; /* linkage unclear by 3.5, since the prior visible
                        declaration has no linkage.

                        external linkage by 7.1.1, since there is no visible
                        name with namespace scope. */
        }
      }
Resolution:
      Change the sentence in 3.5 as follows:
      "The name of a function declared in a block scope or a variable declared
       extern in a block scope has external linkage, unless the declaration
       matches a visible declaration of namespace scope with internal linkage,
       in which case the object or function has internal linkage."
Resolution:
Requestor: Mike Holly
Owner: Josee Lajoie (Linkage)
Emails:
      core-5056
Papers:
. . . . . .

Work Group: Core
Issue Number: 473
Title: linkage of local static variables needs to be better described
Section: 7.1.1 [dcl.stc] Storage class specifiers
Status: active
Description:
3.5 paragraph 7 says:
"Names not covered by these rules have no linkage. Moreover, except as noted a name declared in a local scope (3.3.1) has no linkage."

7.1.1 paragraph 4 says:
"A name declared with a static specifier in a scope other than class scope (3.3.5) has internal linkage."

The latter would seem to imply that both functions in the following translation unit are modifying the same object:

```
void f() {static int x; x++;}
void g() {static int x; x++;}
```

Resolution:
Change 7.1.1 para 4 to say:
"A name declared with a static specifier in a namespace scope has internal linkage, and in a block scope has no linkage."
Requestor: Mike Holly
Owner: Josee Lajoie (Linkage)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 454
Title: Can two inline functions call one another?
Section: 7.1.2 [dcl.fct.spec] Function specifiers
Status: active
Description:
p3 says:
"A call to an inline function shall not precede its definition."

Doesn't this prohibit existing practice?

```
class X {
   void f() { g(); }
   void g() { h(); }
};
```

Resolution:
Proposal:
Replace the sentence above in 7.1.2 p3 with:
"An inline function must be declared inline before it is used."
Requestor: John Max Skaller
Owner: Josee Lajoie (Linkage)
Emails:
core-5061
Papers:

. . . . . .

Work Group: Core
Issue Number: 502
Title: What is the linkage name of unnamed enum types introduced by a typedef?
Section: 7.1.3 [dcl.typedef] The typedef specifier
Status: active
Description:
The rules described in 7.1.3 p5 apply to enumeration types as well.
Resolution:
Proposal:
Change the text in 7.1.3 p5 as follows:

"An unnamed class _or enum_ defined in a declaration with a typedef
specifier gets a dummy name. For linkage purposes only, the first
typedef-name declared by the declaration is used to denote the class
type _or enumeration type_ in the place of the dummy name.
...
The typedef-name is still only a synonym for the dummy name and shall
not be used where a true class name _or enum name_ is required.
...
If an unnamed class _or enum_ is defined in a typedef declaration but
the declaration does not declare a class type _or enumeration type_,
the name of the class for linkage purposes is a dummy name."

And delete paragraph 6, which becomes unnecessary.

Requestor: Bill Gibbons
Owner: Josee Lajoie (Linkage)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 116 (WMM.65)
Title: Is "const class X { };" legal?
Section: 7.1.5 [dcl.type] Type Specifiers
Status: active
Description:
Is "const class X { };" legal, and, if so, what does it mean?
Resolution:
Requestor: Mike Miller
Owner: Steve Adamczyk (Classes)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 503
Title: Better semantics of bitfields of enumeration type needed
Section: 7.2 [dcl.enum] Enumeration declarations
Status: active
Description:
7.2p5 describes the underlying type of enumeration types.
It should be made clear that this description does not apply to
the underlying type of enumeration bit-fields.

Also, something should be said about the signedness of enumeration
types. Bill Gibbons's suggested words:
"Even though the underlying type of an enumeration type will be either
signed or unsigned, enumerations themselves are neither signed nor
unsigned. [For example, a two-bit bit-field can hold an enumeration
with values {0,1,2,3}.]"
Resolution:
Requestor: Bill Gibbons
Owner: Steve Adamczyk (Types)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 474
Title: Lookup of namespace names in alias declarations
Section: 7.3.2 [namespace.alias]
Status: active
Description:
Are names used in namespace-alias-definitions looked up only as
namespace names?

namespace very_long_name { }

```
            void f() {
                  int very_long_name;
                  namespace VLN = very_long_name;  //1
            }

      Does the namespace-alias-definition on line //1 find the namespace
      name very_long_name declared in global scope?
      Or is the declaration in error?
Resolution:
      Proposal:
      When looking up the names mentioned in the qualified-namespace-specifier
      of a namespace-alias-definition, only namespace names are looked up.
Requestor:
Owner:          Steve Adamczyk (Name lookup)
Emails:
Papers:
. . . . . .
Work Group:     Core
Issue Number:   475
Title:          Lookup of namespace names in using directives
Section:        7.3.4 [namespace.udir]
Status:         active
Description:
      Are names used in using directives looked up only as namespace names?

            namespace NS { }

            void f() {
                  int NS;
                  {
                        using namespace NS; //1
                  }
            }

      Does the using directive on line //1 find the namespace name NS in
      global scope?
Resolution:
      Proposal:
        When looking up the names mentioned in using directives, only namespace
        names are looked up.
Requestor:
Owner:          Steve Adamczyk (Name lookup)
Emails:
Papers:
. . . . . .
Work Group:     Core
Issue Number:   78 (also WMM.38)
Title:          Linkage specification and calling protocol
Section:        7.5 [dcl.link] Linkage Specifications
Status:           active
Description:
      extern "C" {
            // Typedef defined in extern "C" blocks:
            // What is the linkage of the function pointed at by 'fp'?
            typedef int (*fp)(int);

            // Type of a function parameter:
            // What is the linkage of the function pointed at by 'fp2'?
            int f(int (*fp2) (int));

            // Can function with C linkage be defined in extern "C" blocks?
                int f2(int i) { return i; }

            // Can static function with C linkage be defined in
```

```
            // extern "C" blocks?
                static int f3(int i) { return i; }
      }
      If function declarations/definitions placed inside the extern "C" block
      have different properties from the ones placed outside these blocks,
      many areas of the C++ language will have to be aware of difference.
      i.e.
      a. function overloading resolution
        b. casting
            one will need to be able to cast from a pointer to a function
            with linkage "X" to a pointer to a function with linkage "Y".
      In short, it needs to be determined to what extent the linkage is part
      of the type system.

      [ JL: ]
            The standard should not force implementations to accept the
            following code:
                  extern "SomeLinkage" int (*ptr)();
                  int (*ptr_CXX)();
                  ptr_CXX = ptr; // 1
            i.e. an implementation should be able to issue an error for
            line (// 1).
Resolution:
      See 94-0034/N0421 for a proposed resolution.
Requestor:      John Armstrong (johna@kurz-ai.com)
Owner:          Josee Lajoie (Linkage)
Emails:
      core-1583, core-1584, core-1585, core-1586, core-1587, core-1589
      core-1590, core-1591, core-1594, core-1595, core-1597, core-1598
      core-1599, core-1608, core-1609, core-1612
        core-920 (Hansen),core-985 (O'Riordan),core-1064 (Miller)
Papers: 94-0034/N0421
. . . . . .
Work Group:     Core Language
Issue Number:   420
Title:          Linkage of C++ entities declared within `extern "C"'.
Section:        7.5 [dcl.link] Linkage Specification
Status:         active
Description:
      Given a declaration or definition of some C++ entity (e.g.  a data
      member, a function member, and overloaded operator, an anonymous union
      object, etc) whose existance within an otherwise standard conforming
      program written in ANSI/ISO C would be a violation of the language
      rules, what is the effect of the linkage specification on the
      declarations/definitions of the C++ specific entities:
      Example:
      extern "C" {
              struct S {
                        int data_member;
                };
                int operator+ (S&, int);
      }
Resolution:
Requestor:      Ron Guilmette
Owner:          Josee Lajoie (Linkage)
Emails:
Papers:
. . . . . .
Work Group:     Core Language
Issue Number:   486
Title:              What is the effect of multiple linkage declarations?
Section:        7.5 [dcl.link] Linkage Specification
Status:         active
Description:
```

The grammar allows multiple linkage specifications, like

```
extern "C" extern "C++" int foo() ;
```

All 7.5[dcl.link]p2 says is "linkage specifications nest."

Do we really want to allow declarations such as the above?
And if we do, we should say what they mean, i.e.
"When linkage specifications nest the innermost one determines the linkage".

Resolution:
Proposal:
Add to 7.5p2 the sentence suggested by Jerry.

Requestor: Jerry Schwarz
Owner: Josee Lajoie (Linkage)
Emails:
Papers:

. . . . . .

Chapter 8 - Declarators

Work Group: Core
Issue Number: 456
Title: What is the linkage of a cv-qualified reference to T?
Section: 8.3.2 [dcl.ref] References
Status: active
Description:

p1 says:
"At all times during the determination of a type, types of the form 'cv-qualified reference to T' is adjusted to be 'reference to T'."

This only says that cv-qualifiers are ignored during type determination.
Aren't cv-qualifiers ignored for other purposes as well (establishing the reference's linkage for example)?
For example,

```
typedef int& IR;
const IR ref = 99;
```

does 'ref' have internal or external linkage?

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

. . . . . .

Work Group: Core
Issue Number: 504
Title: Isn't it ill-formed to directly declare a cv-qualified reference?
Section: 8.3.2 [dcl.ref] References
Status: active
Description:
Isn't the following declaration ill-formed?

```
int & const cri = ...;
```

That is, isn't it ill-formed to directly declare a cv-qualified reference?

Resolution:
Requestor: Bill Gibbons
Owner: Steve Adamczyk (Declarators)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 457
Title: What is the linkage of a cv-qualified array?

Section: 8.3.4 [dcl.array]
Status: active
Description:
        p1 says:
      "At all times during the determination of a type, types of the form
         'cv-qualifier-seq array of N T' is adjusted to be 'array of N
        cv-qualifier-seq T'."

        This only says that cv-qualifiers are ignored during type determination.
      Aren't cv-qualifiers ignored for other purposes as well (establishing
      the array's linkage for example)?
      For example,
            typedef int A[5];
              const A x;
        What is the linkage of x?
esolution:
Requestor:
Owner: Steve Adamczyk (Declarators)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 439
Title: Are there any restrictions on the parameter before the ellipsis?
Section: 8.3.5 [dcl.fct] Functions
Description:
        Can the parameter before the ellipsis be a reference?
Resolution:
Requestor:
Owner: Steve Adamczyk (Declarators)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 482
Title: Are cv-qualifiers allowed in a typedef for a function type?
Section: 8.3.5 [dcl.fct] Functions
Description:
        struct A {
                typedef void FTYPE() const;
                void f() const;
                FTYPE f1;
        };

      Do f and f1 have the same type?
Resolution:
      Bill Gibbons' proposed resolution:

      Since const/volatile qualifiers are part of the type of a member
      function, they should be allowed on typedefs. Of course if that ends
      up putting cv qualification on a non-member, a static member, a
      constructor or a destructor, the program is ill-formed - exactly as if
      it were done without the typedef.

      Bill Gibbons suggested the following change to 8.3.5 [dcl.fct].
      Paragraph 3 presently says:

        "... A cv-qualifier-spec can only be part of a declaration or
         definition of a nonstatic member function, and of a pointer to
         member function; see[class.this]. It is part of the function type."

      He proposed changing it to:

        "... A cv-qualifier-spec can only be part of a declaration or

definition of a nonstatic member function, or part of a
pointer to member function type, or part of a typedef for a
function type; see [class.this]. It is part of the function
type. Typedefs of cv-qualified function types can only be used
to declare nonstatic member functions and to form pointer to member
types."
Requestor: Bill Gibbons
Owner: Steve Adamczyk (Declarators)
Emails:
core-5447
Papers:
. . . . . .
Work Group: Core
Issue Number: 476
Title: Can objects with "indeterminate initial value" be referred to?
Section: 8.5 [dcl.init] Initializers
Description:
8.5p6 says:
"If no initializer is specified for an object with automatic or
dynamic storage duration, the object and its subobjects, if any, have
an indeterminate initial value."

The C standard specifies that accessing a variable with indeterminate
value results in undefined behavior, but the C++ draft contains no such
language.
Also, a definition of "indeterminate value" is needed in the C++
standard.
Resolution:
Add the following text at the ned of 8.5 paragraph 6:
"Referring to an object with an indeterminate value results in
undefined behavior."

For the definition of indeterminate value, Steve suggests using C's
definition.
Requestor: Steve Clamage
Owner: Josee Lajoie (Object Model)
Emails:
Papers:
. . . . . .
==========
Chapter 9 - Classes

Work Group: Core
Issue Number: 514
Title: The list of member types a POD class cannot have is not complete
Section: 9 [class]
Status: active
Description:
p4 says:
"A POD-struct is an aggregate class that has no members of type
reference, pointer to member, non-POD-struct or non-POD-union.
Similarly, a POD-union is an aggregate union that has no members of
type reference, pointer to member, non-POD-struct or non-POD-union."
This forgets arrays of such types.
Resolution:
Replace the sentences above with:
"A POD-struct is an aggregate class that has no members of type
pointer to member, non-POD-struct, non-POD-union (or array of such
types) or reference. Similarly, a POD-union is an aggregate union
that has no members of type pointer to member, non-POD-struct,
non-POD-union (or array of such types) or reference."
Requestor:
Owner: Steve Adamczyk (Types)
Emails:

Papers:
. . . . . .
Work Group: Core
Issue Number: 252
Title: Where can the definition of an incomplete class object appear?
Section: 9.1 [class.name] Class names
Status: active
Description:
must an incomplete class object be completed in the same scope?
9.1p24 In C, a struct-or-union of incomplete type must be completed in the same scope as the incomplete-type declaration, or it remains an incomplete type.
[We believe the same is intended for incompletely-defined classes in C++, but the document is not yet clear enough to tell.]

[Note JL:]
The resolution needs to clarify the following test case as well:

```
class C; //1
union {
        class C { ... }; //2
        ...
};
```

Does line //2 defines the class declared on line //1?
Resolution:
Requestor: Tom Plum / Dan Saks
Owner: Steve Adamczyk (Types)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 479
Title: How can typedefs be used to declared member functions?
Section: 9.2 [class.mem] class members
Status: active
Description:
Is this legal?

```
typedef void voidf(int);
class foo {
      voidf vfunc;          // legal? (unsure)
      static voidf sfunc;   // legal? (I think so)
};
```

The type of vfunc is certainly not voidf.

Another example from Bill Gibbons:

```
struct A {
          typedef void FTYPE();
          void f();
          FTYPE f1;
  };
```

Do f and f1 have the same type?

And how does this affect pointer to members?
More examples from Bill Gibbons:

```
struct T { void f(); };
typedef void F();
F T::*pmf = &T::f;  // pointer to member function ?
```

```
                        // Bill's proposed answer yes.

      struct T { void f(); };
      typedef void (*PF)();
      PF T::pmf = &T::f;  // pointer to member function ?
                        // Bill's proposed answer no.

      That is, once a declarator has been established to be a pointer to
      member, the distinction of pointer to data member or pointer to member
      function is entirely dependent on the type, not the syntax.
Resolution:
Requestor:  Mike Ball
Owner:          Steve Adamczyk (Types)
Emails:
      core-5374, core-5447
Papers:
. . . . . .
Work Group:     Core
Issue Number:     335
Title:          Can unions contain reference members?
Section:        9.6 [class.union] Unions
Status:         active
Description:
      union {
            int *p;
            int &r;
      };
      int i, i;
      p = &i;
      r = 1000;
Resolution:
Requestor:      John Armstrong
Owner:              Steve Adamczyk (Unions)
Emails:
        core-2028
Papers:
. . . . . .
Work Group:     Core
Issue Number:   266
Title:          Access specifiers in union member list
Section:        9.6 [class.union] Unions
Status:         active
Description:
      9.6p3.2 - anonymous union may not have private or protected members.
      This seems to imply that anonymous union may have public members;
      and that non-anonymous union may have any access modifiers.
      Is this wording really what is intended?
Resolution:
Requestor:      Tom Plum / Dan Saks
Owner:            Steve Adamczyk (Unions)
Emails:
Papers:
. . . . . .
Work Group:     Core
Issue Number:   105 (WMM.27)
Title:          How can static members which are anon unions be initialized?
Section:        9.6 [class.union] Unions
Status:         active
Description:
        This is from Mike Miller's list of issues:
        class C {
                static union {
                        int i;
                        char * s;
```

```
                };
                union {
                        const int a, b;
                };
        };
        int C::i = 3;  // ? Is this syntax valid?
        int C::a = 5;  // ? Is this syntax valid?
Resolution:
Requestor:      Mike Miller
Owner:          Steve Adamczyk (Unions)
Emails:
Papers:
. . . . . .
Work Group:     Core
Issue Number:   478
Title:          can a union constructor initialize multiple members?
Section:        9.6 [class.union] Unions
Status:         active
Description:
    can a union constructor initialize multiple members?
Resolution:
Requestor:  Neal Gafter
Owner:          Steve Adamczyk (Unions)
Emails:
Papers:
. . . . . .
Work Group:     Core
Issue Number:   505
Title:          Must anonymous unions declared in unnamed namespaces also be
             declared static?
Section:        9.6 [class.union] Unions
Status:         active
Description:
      9.6p3 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.
Resolution:
      Proposal:
      Replace the sentence above with the following:
      "Anonymous unions declared in a named namespace or in the global
       namespace shall be declared static."
Requestor:  Bill Gibbons
Owner:          Steve Adamczyk (Unions)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number:     47
Title:            enum bitfields - can they be declared with < bits than
required
Section:    9.7 [class.bit] Bitfields
Status:         active
Description:
      enum ee { one, two, three, four };
      struct S {
            ee bit:1; // allowed?
      };
Resolution:
Requestor:      ?
Owner:          Steve Adamczyk (Declarators)
Emails:
      core-1578
```

Papers:
. . . . . .
Work Group: Core
Issue Number: 267
Title: What does "Nor are there any references to bitfields" mean?
Section: 9.7 [class.bit] Bitfields
Status: active
Description:
9.7p3.5: "Nor are there references to bit-fields." Does this actually prohibit anything? A simple attempt to make a reference refer to a bit-field just creates a temporary:

```
union { int bitf:2; } u;
const int & r = u.bitf;
```

Or is this a syntactic restriction that prohibits something like

```
union { int (&rbitf):2 } u;
```

Or is it meant to prohibit the use of typedefs to attempt it, such as

```
union { typedef int bitf_t:2; bitf_t &rbitf; } u;
```

The intent needs clarifying.
Resolution:
Requestor: Tom Plum / Dan Saks
Owner: Steve Adamczyk (Declarators)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 458
Title: When is an enum bitfield signed / unsigned?
Section: 9.7 [class.bit] Bitfields
Status: active
Description:

```
enum Bool { false=0, true=1 };
struct A {
   Bool b:1;
};
A a;
a.b = true;
if (a.b == true) // if this is sign-extended, this fails.
```

Resolution:
Bill Gibbons proposed resolution:
Add after the sentence 9.7p5:
"It is implementation defined whether plain (neither explicitly signed or unsigned) int bitfield is signed or unsigned."
"...; enumeration bit-fields are neither signed nor unsigned."
Requestor: Sam Kendall
Owner: Steve Adamczyk (Declarators)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 525
Title: Implementation specific mapping of bitfields
Section: 9.7 [class.bit] Bitfields
Status: active
Description:
9.7p1 says:
"1 Fields straddle allocation boundaries on some machines and not on others. ... Fields are assigned right-to-left on some machines, left-to-right on others.

2 An unnamed bit-field is useful for padding to conform to externally imposed layouts.
"

These sentences are non-mormative, and should be moved as note or

footnote.
Resolution:
Do as he says ;-)
Requestor: Sean Corfield
Owner: Josee Lajoie (Core Editorial)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 68 (WMM.58)
Title: How do access control apply to members of nested classes in the definition of the owning class?
Section: 9.8 [class.nest] Nested Class Declarations
Status: active
Description:
From Mike Miller's list of issues:
How is access control applied to members of nested classes?

```
class X {
        class Y { enum E { E1, E2 }; };
public:
        Y::E f() { return Y::E1; }      //1
};
```

Can Y::E be accessed as shown on line //1?
Can Y::E1 be accessed as shown on line //1?
The WP does not specify this.
Resolution:
9.8p2 says:
"Member functions of a nested class have no special access to members of an enclosing class; they obey the usual access rules. Member functions of an enclosing class have no special access to members of a nested class; obey the usual access rules."

This should be changed to apply to more than just member functions...
This should be true for the entire scope of the enclosing or nested class. For example:

"The scope of a nested class has no special access to members of an enclosing class and the usual access rules shall be obeyed. The scope of an enclosing class has no special access to members of a nested class and the usual access rules shall be obeyed."
Requestor: Mike Miller
Owner: Steve Adamczyk (Access Specifications)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 27
Title: What is the access of nested class types declared multiple times
in the owning class definition?
Section: 9.8 [class.nest] Nested Class Declarations
Status: active
Description:

```
struct ss {
public:
        struct s;
        union u;
        class c;
protected:
        struct s;
        union u;
        class c;
private:
        struct s;
```

```
            union u;
            class c;
      };
      What is the accessibility of s, u, c?
Resolution:
Requestor:  Ron Guilmette
Owner:          Steve Adamczyk (Access Specifications)
Emails:
      core-1517
Papers:
. . . . . .
================================================================================
 Chapter 10 - Derived classes
------------------------------
Work Group:     Core
Issue Number:   441
Title:          In which scope is the base class clause looked up?
Section:        10 [class.derived] Derived classes
Status:         active
Description:
        class C {
          class A { };
          class B : A { }; //1
        };
      Is A looked up in the scope of C or in the scope of B?
        Is the declaration on line //1 ill-formed because the nested class B
      cannot refer to the private type A declared in C?
      Or is it well-formed because the name A can be used in the scope of C?
Resolution:
Requestor:
Owner:          Steve Adamczyk (Name Lookup)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number:     11
Title:            How do retrictions on member mapping apply with multiple
                inheritance?
Sections:   10.1 [class.mi] Multiple Base Classes
Status:         active
Description:
      10.1 paragraph 2 says that the order of derivation may be significant
        to the storage layout. How do the restrictions described in 9.2
        paragraph 11 relate to members declared in base classes?
Resolution:
Requestor:  Scott Turner
Owner:          Josee Lajoie (Object Model)
Emails:
      core-1456
Papers:
. . . . . .
Work Group:     Core
Issue Number:   442
Title:          Can a class have a direct and an indirect base of the same type?
Sections:       10.1 [class.mi] Multiple base classes
Status:         active
Description:
        class A { };
        class B : public A { };
        class C : public A, public B { };

        Is this allowed?
        Since class A's members can never be referred to, can an implementation
        optimize the mapping for class C and not map the direct base class A?
```

CFRONT omitted mapping the direct base class A.

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

. . . . . .

Work Group: Core
Issue Number: 481
Title: Can two base class subobjects be allocated at the same address?
Sections: 10.1 [class.mi] Multiple base classes
Status: active
Description:

```
struct B { void f(); };
struct L : B{};
struct R : B{};
struct D : L,R{};
```

Since B has no data members, can B have the same address as another member subobject of of D?
That is, can a base class subobject have zero size?

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

. . . . . .

Work Group: Core
Issue Number: 446
Title: Can explicit qualification be used for base class navigation?
Sections: 10.1 [class.mi] Multiple base classes
Status: active
Description:

Can explicit qualification be used for base class sublattice navigation?

```
class A {
public:
  int i;
};
class B : public A { };
class C : public B { };
class D {
public:
  int i;
};
class E : public D { };
class F : public E { };
class Z : public C, public F { };
Z z;
... z.F::E::D::i; // is qualification allowed here to navigate the base
                  // class sublattice?
```

Resolution:
Requestor: Bill Gibbons
Owner: Steve Adamczyk (Name Lookup)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 447
Title: When should a class without a final overrider be ill-formed?
Sections: 10.3 [class.virtual] Virtual Functions
Status: active
Description:

```
        p7 shows:
        struct A {
           virtual void f();
        };
        struct B : virtual A {
           void f();
        };
        struct C : virtual A {
           void f();
        };
        struct E : B, C { };   //1

        Is the declaration of E on line //1 ill-formed?
        Why not wait until an object of class E is created to issue the error
        message?
        Making the declaration of E on line //1 ill-formed prevents programs
        to further derive from E and override the virtual member function.
        i.e.
        struct F : E {
           void f();
        };
Resolution:
Requestor:
Owner:          Josee Lajoie (Object Model)
Emails:
Papers:
. . . . . .
================================================================================
 Chapter 11 - Member Access Control
----------------------------------
Work Group: Core
Issue Number:     22
Title:            Must implementations respect access restrictions?
Section:    11.1 [class.access.spec] Access Specifiers
Status:         active
Description:
      What are the access restrictions used for?
      Are they only limitations on what the programmer may write?
      Must implementations also respect them for the compiler generated code?
      (i.e. use of default and copy constructors, use of destructors at
        the end of a block or at the end of the program).

        And if implementations must respect access restriction, when must it
        report the errors?

        struct B {
        private:
            ~B () { }
        };

        struct D : public B {
           ~D () { }  // is the mere existence of D::~D considered an
                      // implicit call of B::~B and therefore a protection
                      // violation?
        };

        void f() {
           D d;       // Or is this an error?
         }            // Or is the error when d is destroyed?

Resolution:
Requestor:  Jerry Schwarz
Owner:          Josee Lajoie (Object Model)
Emails:
```

core-1525

Papers:

. . . . . .

Work Group: Core
Issue Number: 284
Title: access to base class ctor/dtor
Sections: 11.2 [class.access.base] Access Specifiers for Base Classes
Status: active
Description:

base access rules don't apply to non-inherited members
11.2 After much discussion, we believe that the intent is as follows: "The members whose accessibility is described in this section are only those members which are inherited from the base class; the non-inherited members such as constructors, destructors, and assignment operators are not subject to these accessibility rules."

I still have problems with this one. There are two separate issues, one minor and one substantive. Minor issue: The ctor, dtor, and assignment op of a base are not accessible as "members of the derived class" (because they aren't inherited in the derived class), but I believe that access restrictions still apply to them. So it's just a job of re-wording 11.2/p1 to clarify this.

But the substantive issue is a real problem. The vagueness has been unresolved so long that implementations have differed in the treatment of access declarations upon base class ctors and dtors. Could you discuss this with me and we'll determine how to proceed.

Resolution:
Requestor: Tom Plum / Dan Saks
Owner: Josee Lajoie (Object Model)
Emails:
Papers:

. . . . . .

Work Group: Core Language
Issue Number: 388
Title: Access Declarations and qualified ids
Section: 11.3 [class.access.dcl] Access Declarations
Status: active
Description:

The section says:
The base class member is given, in the derived class, the access in effect in the derived class declaration at the point of the access declaration.

It isn't clear to me what this means for

```
class B { public: int i ; } ;
class D : private B {
   B::i ;
};

D* p ;
p->i ;  // clearly legal
p->B::i ;
```

I don't care strongly about this, but I think it should be clarified. (And added as an example).

Resolution:
Requestor: Jerry Schwarz
Owner: Steve Adamczyk (Access Specifications)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 448

Title: Can '::' be used to declare global functions as friends?
Section: 11.4 [class.friend] Friends
Status: active
Description:
Should it be allowed to prefix the declarator of a friend function declaration with '::' to indicate that a global function is the friend?

```
void f();
class C {
   void f();
   friend void ::f();
 };

or

class B {
   void f();
 };
class C {
   class B { };
   friend void ::B::f();
};
```

Resolution:
Requestor:
Owner: Steve Adamczyk (Name Lookup)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 515
Title: How can friend classes use private and protected names?
Section: 11.4 [class.friend] Friends
Status: active
Description:
11.4 p2 says:
"Declaring a class to be a friend implies that private and protected names from the class granting friendship can be used in the class receiving it."

This is not very explicit.
Where can the private and protected names be used in the befriended class?
In the base classes of the befriended class?
In the nested classes of the befriended class?

Resolution:
Requestor: Erwin Unruh
Owner: Steve Adamczyk (Name Lookup)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 449
Title: restriction on protected member access should not apply to types, static members, and member constants
Section: 11.5 [class.protected] Protected member access
Status: active
Description:
P1 of 11.5 now implies that:

```
class C {
protected:
   int i;
   static int j;
```

```
};
class D : public C {
   void f();
};
void D::f() {
   C::i = 4;  //OK, implied 'this' pointer, qualification ignored
   C::j = 6;  //error, nested-name-specifier must represent the
              //       derived class
}
```

The rules for static and nonstatic members should be as uniform as possible. In particular, changing a member from static to non-static or vice-versa should have no effect on the whether the program violates any access rules.

But this isn't so in the proposed set of rules.

Since we don't want to break the "implicit this" rules, I think the only solution is to allow friends and members of the derived class to access static members of the base class with a base class qualifier. That is, apply the access rules as if the member were nonstatic. This change would make the second line in the above example well-form

---

Steve Adamczyk in core-5598:
11.5 [class.protected] says:

> "A friend or a member function of a derived class can access a protected static member, type or enumerator constant of a base class; if the access is through a qualified-id, the nested-name-specifier must name the derived class (or any class derived from that class)."

I remember that when the wording for the qualified-id case was added a few years back, the issue being addressed was pointers-to-members (exclusively).

Where it used to be clear in the ARM that this additional protected member access check applied only to nonstatic members, and therefore not to types, member constants, and static members, now the wording has made it clear that the check IS done for those.

If you read the commentary in the ARM, you see why this check is useful, and the problems being prevented don't apply to static members etc. I think this is just a mistake that's spreading.

Resolution:
Requestor: Bill Gibbons
Owner: Steve Adamczyk (Access Specifications)
Emails:
Papers:

. . . . . .

Chapter 12 - Special Member functions

Work Group: Core
Issue Number: 379
Title: Invoking member functions which are "not inherited".
Section: 12.1 [class.ctor] Constructors
12.4 [class.dtor] Destructors
12.8 [class.copy] Class Copy
Status: active
Description:

Section 5.1/8 of the 1/93 working paper says:

> "A nested-class-specifier (9.1) followed by :: and the name of a member of that class (9.2) or a member of a base of that class (10) is a qualified-id; its type is the type of the member. The

```
                result is the member."

        Section 12.1/1 says:
            "Constructors are not inherited."

        Section 12.4/6 says:
                "Destructors are not inherited."

        May a member of a given base class type which is "not inherited" by
        another class type (derived from the given base class type) be invoked
        for an object whose static type is the derived class type if the
        invocation is done using the class-qualified name syntax?  If, not, is
        an implementation obliged to issue a compile-time diagnostic for such
        usage?

        Is the behavior "well defined" if an attempt is made to invoke a
        non-inherited member for an object whose static type is that of the base
        class but whose dynamic type is that of the derived class?
                struct B {
                        virtual ~B () { }
                };

                struct D : public B {
                        ~D () { }
                };

                D D_object;
                D D_object2;
                B *B_ptr = &D_object2;

                void caller ()
                {
                        D_object.B::~B();               // ok?
                        B_ptr->~B();                    // ok?
                }
Resolution:
Requestor:      Ron Guilmette
Owner:          Josee Lajoie (Object model)
Emails:
Papers:
        94-0193R1/N0580
. . . . . .
Work Group:     Core
Issue Number:   477
Title:          When can an implementation create temporaries?
Section:        12.2 [class.temporary]
Status:         active
Description:
      12.2[class.temporary]/2 still contains the phrase
      "Precisely when such temporaries are introduced is
      implementation-defined".

      This seems extremely vague.  For example, is the implementation allows
      to introduce a temporary in the following example?

      class Foo { };
      int main() {
            return 0;
      }

      Can the temporary be of type `Foo'?
      If `Foo' had a private constructor, could the implementation introduce
      a temporary of type `Foo', and then reject the program as ill-formed??
```

(Perhaps the answers to these questions are obvious, but I don't think they are obvious from the text in the working paper.)

Resolution:
Requestor: Fergus Henderson
Owner: (Temporaries)
Emails:
core-5542
Papers:

. . . . . .

Work Group: Core
Issue Number: 516
Title: What is the lifetime of "helping" temporaries?
Section: 12.2 [class.temporary]
Status: active
Description:

12.2p4 says:
"The first context is when an expression appears as an initializer for a declarator defining an object. In that context, the temporary that holds the result of the expression shall persist until the object's intialization is complete."

All temporaries created in the evaluation of the expression should last until the object is initialized. It should NOT be limited to the result temporary. For example:

```
char *a =  string().c_str;
```

12.2p5 says:
"The temporary bound to a reference or the temporary containing the sub-object that is bound to the reference persists for the lifetime of the reference initialized..."

It should be made explicit, whether helping references also last as long as the reference.

```
char * &ra = string().c_str;
```

Does the string temporary last as long as the reference?

Resolution:
Requestor: Erwin Unruh
Owner: (Temporaries)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 507
Title: Must a temporary initializing an object be destroyed _as soon as_ it has been copied?
Section: 12.2 [class.temporary]
Status: active
Description:

12.2p4 says:
"The object is initialized from a copy of the temporary; during this copying, an implementation can call the copy constructor many times; the temporary is destroyed as soon as it has been copied."

Bill says:
don't recall the requirement of "as soon as it has been copied"; I thought it was any time before the end of the initialization.

Resolution:

Proposal:
Change the sentence above to:
"... the temporary shall be destroyed after it has been copied, before or when the initialization completes."

Requestor: Bill Gibbons
Owner: (Temporaries)
Emails:

Papers:
. . . . . .
Work Group: Core Language
Issue Number: 509
Title: How does "destroy in reverse order of creation" work for temporaries bound to references vs temporaries destroyed at the end of the initialization?
Section: 12.2 [class.temporary]
Status: active
Description:
12.2p6:
"In all cases, temporaries are destroyed in reverse order of creation."

There is an exception to "reverse order of creation":

> Temporaries are destroyed in reverse order of creation, except those temporaries created in an expression used to initialize a reference; these are destroyed when the reference is destroyed, in reverse order of creation.

Resolution:
Proposal:
Add at the end of paragraph 4:
"If many temporaries are created during an object's initialization, these temporaries are destroyed in reverse order of creation."

Add at the end of paragraph 5:
"Temporaries bound to references are destroyed in reverse order of creation."

Delete paragraph 6.
Requestor: Bill Gibbons
Owner: (Temporaries)
Emails:
Papers:
. . . . . .
Work Group: Core Language
Issue Number: 347
Title: Limitations on declarations of user-defined type-conversions.
Section: 12.3.2 [class.conv.fct] Conversion functions
Status: active
Description:
Given a declaration such as:

```
struct S {
   operator T ();
};
```

... where `T' is a typedef name declared in an earlier typedef declaration, must a standard conforming implementation issue a diagnostic for the declaration of the type conversion operator (as shown above) if, at the point of declaration of the type conversion operator itself, the type T is:

- A complete array type?
- An incomplete array type?
- An incomplete class type?
- A function type?

Resolution:
Requestor: Ron Guilmette
Owner: Steve Adamczyk (Type Conversions)
Emails:
Papers:
. . . . . .
Work Group: Core Language
Issue Number: 485
Title: Order of destruction of base classes and members
Section: 12.4 [class.dtor] Destructors

Status: active
Description:
12.4p5 says:
"Bases and members are destroyed in reverse order of their construction."
Since construction takes place over time, is the order that is to be reversed the one when the constructor begins or when it finishes.
Resolution:
Jerry's proposed answer:
The one when it finishes.
Requestor: Jerry Schwarz
Owner: Josee Lajoie (Destruction)
Emails:
core-5596
Papers:

. . . . . .

Work Group: Core
Issue Number: 293
Title: Clarify the meaning of y.~Y
Section: 12.4 [class.dtor] Destructors
Status: active
Description:
Resolution:
12.4p22 The notation y.~Y() is explicitly approved of by the example at bottom of ARM page 279), but nothing in the draft gives this explicit approval. Implementations differ. Committee should approve it or disapprove it.
Requestor: Tom Plum / Dan Saks
Owner: Josee Lajoie (Destruction)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 508
Title: When is a destructor implicitly-defined?
Section: 12.4 [class.dtor] Destructors
Status: active
Description:
12.4p4 says:
"An implicitly-declared destructor is implicitly defined when it is used to destroy an object of its class type."
When is a destructor used?
Resolution:
add a footnote to say:
"Destructors are called implicitly not only when an object of automatic storage duration goes out of scope, or at the end of a program for objects with static storage duration, but also in several situations due to the handling of exceptions."
Requestor: Bill Gibbons
Owner: Josee Lajoie (Destruction)
Emails:
Papers:

. . . . . .

Work Group: Core
Issue Number: 450
Title: How is a class operator new/delete looked up?
Section: 12.5 [class.free] Free Store
Status: active
Description:
[class.free] says:

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

Does that mean that operator new is looked up as a member of T and then
in the current scope, or that it is looked up as a member of T and then
in the enclosing scope of T? In other words, is this code well-formed?

```
struct A { };
struct B {
   char buf[sizeof (A)];
     void *operator new (size_t, void *p, int) { return p; }
     B() { new (buf, 1) A; }
};
```

How about this?

```
namespace foo {
   void *operator new (size_t, void *p, int) { return p; }
   struct A { };
}

namespace bar {
   struct B {
      char buf[sizeof (foo::A)];
      B() { new (buf, 1) foo::A; }
   };
}
```

Resolution:

Neal Gafter answered this issue by saying:
(see 95-0063R1/N0663R1)
"This is already answered by the current WP.
Specifically, operator new is looked up in the scope of the class,
then globally (WP 12.5 [class.free] p2). Operator delete is looked up
both from the point of the delete expression (p7) and from the
destructor(p10)."

[Note JL:]
First, I believe the wording in 12.5 could be made clearer to say
"looked up in the class scope and then globally".

Second, in the light of the new function overload resolution rules
which say that an operator is looked up in a namespace scope N if
it is used with an operand of type class C declared in namespace N,
(see 13.3.1.2 [over.match,oper] p3 and 4), do we want this to be true
for new and delete operators (with placement) as well?
i.e. if an operator new with placement is used to create an object
of type C and C is defined in namescope N, will the scope of namespace N
be searched to find an appropriate operator new with placement?
i.e. do we want to make Jason's 2nd example work?

If we want this to be true, then I believe the wording in 12.5 needs
work.

Requestor: Jason Merrill
Owner: Steve Adamczyk (Name lookup)
Emails:
core-4749
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: active
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
```

Resolution:
Requestor: Mike Miller / Martin O'Riordan
Owner: Josee Lajoie (Construction)
Emails:
core-668
Papers:

. . . . . .

Work Group: Core Language
Issue Number: 359
Title: Timing of Evaluations in Base and Member Initializations
Section: 12.6.2 [class.base.init] Initializing Bases and Members
Status: active
Description:

Section 12.6.2 describes the order in which the members and base class parts of an object of class type are initialized. It does not, however, specify when the expressions used in the "mem-initializers" are to be evaluated.

Consider this example:

```
struct S {
   int i;
   int j;
   S() : i(0), j(i) {}
};
```

12.6.2/1 requires that i be initialized before j. But it seems to permit this order of execution:

- Fetch the (uninitialized) value of i
- Initialize i to 0
- Initialize j to the previously fetched value of i

which is probably not what the programmer intended.

Here, there is no function call (constructor call) to initialize the member i, is there none-the-less a sequence point after i is initialized?

I would suggest adding words similar to:

> The expressions in the expression-list part of a mem-initializer are evaluated (including all side effects) immediately before the corresponding initialization is performed. When one base class or member is initialized before another base class or member, the expressions used in the mem-initializer for the second are evaluated (including all side effects) after the first initialization is performed.

Resolution:
Requestor: Patrick Smith

Owner: Josee Lajoie (Construction)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 95
Title: Volatility, copy constructors, and assignment operators.
Section: 12.8 [class.copy]
Status: active
Description:
It appears that volatile qualification has been overlooked in the specification of copy constructors and assignment operators.
Section 12.1p5 of the WP says:
"A copy constructor for a class X is a constructor whose first argument is of type X& or const X&..."

But a user should be able to pass volatile objects (by reference) to copy constructors and/or assignment operators.

In such cases it would be useful (and symmetric with const qualification) if:
(a) these objects could be used as arguments to copy constructors and assignment operators, and
(b) if the volatility associated with the types of the objects were preserved in the process.
Resolution:
Requestor: Ron Guilmette
Owner: Josee Lajoie (Class Copy)
Emails:
core-1653
Papers:
See paper 95-0056/N0656
. . . . . .
==========
Chapter 13 - Overloading

Work Group: Core
Issue Number: 451
Title: Description of a call to a member function through a pointer to member is missing
Section: 13.2.1.1.1 [over.call.func] Call to named function
Status: active
Description:
Should this section also describe calls to member functions using the pointer to member syntax (.*, ->*) ?
Resolution:
Requestor:
Owner: Steve Adamczyk (function overload resolution)
Emails:
Papers:
. . . . . .
Work Group: Core
Issue Number: 517
Title: Candidate operator functions for built-in operators do not cover the type bool appropriately
Section: 13.6 [over.built] Built-in operators
Status: active
Description:
The type bool is not sufficiently represented. This is most visible for ++ and --, which is different for type bool.
Resolution:
Requestor: Erwin Unruh
Owner: Steve Adamczyk (function overload resolution)
Emails:

Papers:

. . . . . .

Work Group: Core

Issue Number: 518

Title: The candidate operator function for built-in operator= for pointer to members is missing

Section: 13.6 [over.built] Built-in operators

Status: active

Description:

The candidate operator function for built-in operator= for pointer to members is missing.

Resolution:

Requestor: Erwin Unruh

Owner: Steve Adamczyk (function overload resolution)

Emails:

Papers:

. . . . . .