Doc No: X3J16/97-0086 WG21/N1124
Date: Sept 30th, 1997
Project: Programming Language C++
Ref Doc:
Reply to: Josee Lajoie
(josee@vnet.ibm.com)

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

The following core issues were closed at the London meeting either because the Core WG decided to take no action, or because a resolution was adopted by the committee.

```
1.1 [intro.scope]:
  604: Should the C++ standard talk about features in C++ prior to 1985?
1.7 [intro.compliance]:
  602: Clarify the WP conformance model
1.8 [intro.execution]:
  848: What can be done in a signal handler?
2.3[lex.trigraph]:
  744: Is the description of trigraph processing wrong?
2.10 [lex.name]:
  849: Which names are reserved to implementations?
3.2 [basic.def.odr]:
  789: When is a name used in a default argument considered "used"?
3.3.6 [basic.scope.class]:
  664: When does the reevaluation rule for class scope name lookup require a
       diagnostic?
3.4 [basic.lookup]:
  869: Is a class name inserted in its own class scope considered a member
       name for the purpose of name look up?
3.4.2 [basic.lookup.koenig]:
  686: Where is a function name looked up if an argument type is introduced
       with a using-declaration?
  790: What is the associated namespace if the argument has function type?
  791: Does a function declaration need to be visible at the point of the
       call for a function call to be well-formed?
3.4.3 [basic.lookup.qual]:
  665: In X::~Y is Y looked up in the context of the current expression?
3.5 [basic.link]:
  792: What are the rules used to determine whether expressions involving
       nontype template parameters are equivalent?
3.6.1 [basic.start.main]:
  851: Must a diagnostic be issued if main is called in a program?
3.6.2 [basic.start.init]:
  746: What is the order of initialization of a class static data member?
  747: The term "static initialization" needs to be defined
3.6.3 [basic.start.term]:
  852: Should the destruction of array objects be inter-leaved with calls
       to the functions registered with atexit?
3.9.3 [basic.type.qualifier]:
  772: Wording needs to acknowledge there is no such thing as a const
       reference
4.2 [conv.array]:
  885: Can array rvalues decay to pointers?
  773: When is the conversion array of const char to pointer to char applied
       on a string literal?
4.10 [conv.ptr]:
  793: Is it "null pointer constant" or "null-pointer constant"?
  854: Must a null pointer constant be an rvalue of integer type of value 0?
5.1 [expr.prim]:
  855: ::name is not a qualified-id
5.2.2[expr.call]:
```

```
  794: Are recursive calls to main() allowed?
5.2.4[expr.pseudo]:
  795: Should a pseudo-destructor call allow the object expression to have a
       different cv-qualification from the type-name naming the destructor?
5.2.9 [expr.static.cast]:
  774: Should the WP say that converting from void* to original pointer type
       yields a pointer value equal to original pointer?
  775: Is a conversion between a pointer to a struct and a pointer to the
       first member of the struct a static_cast?
5.2.10 [expr.reinterpret.cast]:
  858: Can an expression of any type be cast to its own type using a
       reinterpret_cast?
5.2.11 [expr.const.cast]:
  796: Can a const_cast cast _any_ type to its own type?
5.3.1 [expr.unary.op]:
  860: Is ptr->~T() a call to a destructor?
5.3.4 [expr.new]:
  886: What arguments are passed to placement operator delete?
  669: semantics for new and delete expressions should be separated from the
       requirements for operator new and delete
  797: Is initialization performed if the nothrow operator new returns a null
       pointer value?
  753: Is 'new char[size]' aligned properly to hold an object of any type T?
5.6 [expr.mul]:
  719: Is unsigned arithmetic modulo 2~N for multiplication as well?
5.7 [expr.add]:
  798: What are the semantics of pointer +/- enum?
5.9 [expr.rel]:
  721: Comparisons of pointer to class members need fine tuning
  799: An example illustrating comparisons of pointers to different types and
       different cv-qualifications is needed
5.10 [expr.eq]:
  861: Should the WP say that &x == &y is false if x not same object as y?
5.19 [expr.const]:
  722: The definition of address constant expression needs fine tuning
7.1.1 [dcl.stc]:
  800: Mistake in description of when an incomplete class can be used
7.1.5.1 [dcl.type.cv]:
  862: A local name declared const does not have internal linkage
7.2 [dcl.enum]
  683: What is the underlying type of an enumeration type if the value of an
       enumerator uses the value of a previous enumerator?
7.3.3 [namespace.udecl]:
  672: using-declarations and base class assignment operators
  863: Can the name introduced by a using-declaration be the same as the
       name of an entity already declared in that scope?
  801: Clarification of the interaction of partial specializations and
       using-declarations
  802: Clarification of conversion template instance names and
       using-declarations
7.5 [dcl.link]:
  729: Must extern "C" functions declared in a namespace and a global extern
       "C" function have different signatures and return types?
  749: Can a declaration specify both a storage class and a linkage
       specification?
  750: To which declarator in a member function declaration does the
       extern "C" specifier apply?
8.3.6 [dcl.fct.default]:
  776: Name look up in default argument expressions
8.4 [dcl.fct.def]:
  865: What is the potential scope of a function parameter?
8.5 [dcl.init]:
  751: Should { } be allowed around an initializer that is a string?
  867: copy constructors do not have parameters of derived class type
```

8.5.1 [dcl.init.aggr]:
868: description of aggregate initialization should refer to default initialization
8.5.3[dcl.init.ref]:
804: Can a reference bind directly to what a function call returns if the function returns a reference?
9[class]:
805: Can a zero-size class contain static members, member functions and nested types?
9.4.2 [class.static.data]:
870: Is an error required if a static data member is used and not defined?
9.5 [class.union]:
505: Must anonymous unions declared in unnamed namespaces also be static?
871: Can a class with a constructor but with no default constructor be a member of a union?
10.1 [class.mi]:
624: class with direct and indirect class of the same type: how can the base class members be referred to?
11[access]:
806: 11 para 1 does not cover all members that can refer to the private and protected members of a class
11.5 [class.protected]:
752: When accessing a base class member, the qualification is not ignored
11.8[class.access.nest]:
807: Can local classes within member functions refer to the private members of the member function's class?
12.1[class.ctor]:
808: During the construction of a const object, what happens if the object is modified, and a pointer to const type assumes that the object remains unchanged?
12.2 [class.temporary]:
777: Should it be mentionned in 12.2 that the exception object has a lifetime longer than the full-expression?
874: Clarify lifetime of temporary example
12.4 [class.dtor]:
809: It should be made clear that when the destructor for a derived class implicitly calls the destructor for a base class, the virtual function mechanism is not used
12.6.2 [class.base.init]:
810: When a class has a member and a base class with the same name what does a mem-initializer-id referring to this name designate, the base or the member?
12.6.2 [class.base.init]:
875: If a constructor has no ctor-initializer, but the class has a const member, is the constructor definition ill-formed?
12.8 [class.copy]:
811: Can a base class copy assignment operator that is virtual be overriden by an assignment operator declared in a derived class?
876a:The optimization that allows a copy of a class object to alias another object is too permissive
13.3.1[over.match.funcs]:
778: How does the implicit argument match the implicit parameter of a base class static member function?
13.3.1.2[over.match.oper]:
812: Is the built-in operator for , & -> used if overload resolution is ambiguous?
13.3 [over.match]:
877: 13.3.1.6 isn't about binding to a temporary
13.3.3 [over.match.best]:
813: The partial ordering rules for function templates are overly restrictive
13.3.3.1 [over.best.ics]:
733: Implicit conversion sequences and scalar types

```
13.3.3.2[over.ics.rank]:
  779: identity conversion is preferred over lvalue-to-rvalue conversion
13.6 [over.built]:
  682: operator ?: and operands of enumeration types
  734: ambiguity in "bool & ? void *& : classType&" where classType has an
       operator void*&
  756: most uses of built-in "?" with class operands are ambiguous
14 [temp]
  780: The definition of 'template-declaration' is incomplete
  757: Can a template member function be overloaded?
  814: The semantics of the keyword "export" need to be clarified
  878: Can a template declaration not followed by a definition specify
       export?
  803: The restrictions on default arguments in templates are not
       sufficiently complete
  730a:When are default arguments for member functions of template classes
       semantically checked?
14.1 [temp.param]:
  815: Does the type of a template nontype parameter of array/function type
       decay?
14.2[temp.names]:
  816: There is an ambiguity on ">" with expressions written as default
       arguments
  883: Can "template" be used to specify that an unqualified function
       name refers to a template specialization?
14.3 [temp.arg]:
  758: Can an array name be a template argument?
  759: Initializing a template reference parameter with an argument of a
       derived class type needs to be described
  760: Is a template argument that is a private nested type accessible in
       the template instantiation context?
  782: Can a value of enumeration type be used as a template non-type
       argument?
  879: What conversions can apply to a template argument to bring it to
       the type of the corresponding nontype template parameter?
14.5.1.1 [temp.mem.func]:
  761: Can the member function of a class template be virtual?
14.5.3[temp.friend]:
  817: Clarification of the interaction of friend declarations and partial
       specializations
  818: Friends classes are not well covered in 14.5.3
  880: When does a friend declaration refer to a global function or to a
       template instantiation?
14.5.4[temp.class.spec]:
  819: Were are partial specialization allowed?
  820: Clarification of nontype dependency rules in partial specializations
  821: The restrictions on partial specializations based on the dependency of
       arguments on other arguments are too severe
  881: What class-key can be used in declarations of specializations
       and partial specializations?
14.5.4.2[temp.class.order]:
  822: Clarification of ordering rules for nontype arguments in partial
       specializations
  823: Interaction of partial ordering with default arguments and ellipsis
       parameters
  824: In which contexts should partial ordering of function templates be
       performed?
14.5.4.3[temp.class.spec.mfunc]:
  825: Clarification of rules for partial specializations of member class
       templates
14.5.5.1[temp.arg]:
  762: How can function templates be overloaded?
14.5.5.2 [temp.func.order]:
  763: Partial Specialization: the transformation also affects the function
```

return type

14.6 [temp.res]:

736: How can/must typename be used?

764: undeclared name in template definition should be an error

826: Does the "template" keyword apply to function and static data member templates?

14.6.1[temp.local]:

766: How do template parameter names interfere with names in nested namespace definitions?

827: C is not equivalent to C<T> when C is qualified

14.6.2 [temp.dep]:

784: The examples in 14.6.2 on dependent names need work

828: In what contexts is the use of a qualifier to look in the current template a special case not subject to the usual dependent type restrictions?

829: 14.6.2 para 5 should not only apply when a base class is a template parameter but also when it is a dependent type

14.6.3 [temp.nondep]:

884: no diagnostics required for semantics errors in template definitions

14.6.4.1 [temp.point]:

767: Where should the point of instantiation of class templates be discussed?

830: Are the rules describing the point of instantiation of a function templates too complex?

14.6.4.2[temp.dep.candidate]:

831: Should candidate functions without external linkage in other translation units render a call ill-formed?

14.6.5 [temp.inject]:

832: Difference between the rules in 14.6.5 and 3.4.2 regarding friend function name look up

14.7 [temp.spec]:

833: The definition of "specialization" for member templates is missing

14.7.1 [temp.inst]:

834: Does "delete ap;", where ap's type is a template specialization, cause the template to be instantiated?

835: Does the instantiation of a class template cause the instantiation of the class static data members?

14.7.2 [temp.explicit]:

786: The description of explicit instantiation does not allow the explicit instantiation of members of class templates (including member functions and static data members)

836: What is the point of instantiation for a specialization to which an explicit instantiation directive applies?

837: When can an empty template argument list "<>" be omitted?

14.7.3 [temp.expl.spec]:

787: Make it clear that a user must provide a definition for an explicitly specialized template; if not, the program is ill-formed

838: Does an explicit instantiation directive affect the compilation model for the specified instance?

840: Does the prohibition on default arguments in the definition of a specialization prohibits them in the declarations of member functions of a class specialization?

14.8.1 [temp.arg.explicit]:

841: Are explicit template arguments only allowed in function calls?

14.8.2 [temp.deduct]:

677: Should the text on argument deduction be moved to a subclause discussing both function templates and class template partial specializations?

768: typename keyword missing in some examples

842: Template argument deduction rules for template conversion functions are missing

15[except]:

843: Are "recursive" exceptions allowed?

15.1[except.throw]:

```
  844: Does a rethrow creates a new exception?
  845: If a string literal is thrown, what handler can catch it?
  846: Where does the search for a handler starts if a handler throws an
       exception?
15.2 [except.dtor]:
  769: Are the base class dtors called if the derived dtor throws an
       exception?
15.3 [except.handle]:
  788: Is it implementation defined whether the stack is unwound before
       calling terminate in all of the 8 situations described in 15.5.1?
Annex C:
  680: Annex C subclause C.1 is out of date
  743: Some anachronisms are missing from annex C
Annex E [extendid]:
  891: The list of hexadecimal code for CJK Unified Ideographs seems
       incorrect

==============================================================================
 Chapter 1 - Introduction
--------------------------
Work Group:     Core
Issue Number:   604
Title:          Should the C++ standard talk about features in C++ prior to
                1985?
Section:        1.1 [intro.scope]
Status:         closed
Description:
        UK issue 229:
        "Delete the last sentence of 1.1 and Annex C.1.2. This is the first
         standard for C++, what happened prior to 1985 is not relevant to
         this document."
Resolution:
        At the Nashua meeting, the C compatibility WG decided:
        "Delete references to C.1. Annex C.1 needs to be removed."
Requestor:      UK issue 229
Owner:          (C Compatibility)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   602
Title:          Clarify the WP conformance model
Section:        1.3 [intro.compliance]
Status:         closed
Description:
        Part 1 (resolved):
        o Resolve the inconsistencies in the WP.

          Proposed Resolution:
          Subclause 1.3 should:
          - recognize that some syntactic errors do not require diagnostics,
            either because they are explicitly so described or because they
            are described as resulting in undefined behavior.
          - decouple the requirement to issue a diagnostic from the various
            taxonomies (compile-time vs runtime errors, well-formed vs
            ill-formed programs) and simply require that violations of
            diagnosable rules result in a diagnostic.
          - decouple the requirement to accept and correctly execute programs
            from the various taxonomies and simply require that
            implementations accept and correctly execute programs that
            contain no errors.

        The proposed wording is in Mike Miller's paper.
```

```
        Part 2 (active):
        o Refining the definition of "well-formed" and "ill-formed"

          Clarify that well-formed programs contain no compile-time or
          link-time errors.

          Mike Miller's paper proposes wording to address this issue.
          At the Nashua meeting, the core WG did not agree on whether this
          is a problem that needs to be resolved or whether Mike's proposed
          resolution was acceptable.
Resolution:
        Part 1 was adopted by the committee. The changes are as proposed in
        Mike's paper 97-0023/N1061. Part 2 was rejected.
Requestor:      Mike Miller
Owner:          Josee Lajoie (Conformance Model)
Emails:
Papers:
        97-0023/N1061 Defining Conformance, Rev. 1 by Mike Miller
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   848
Title:          What can be done in a signal handler?
Section:        1.8 [intro.execution]
Status:         closed
Description:
        [Erwin Unruh:]
          Throwing an exception from within a signal handler should be
          undefined.  All you can portably do within a handler is to set
          a global flag of type "volatile sig_atomic_t".

        [Greg Colvin:]
          The C standard allows a signal handler to call signal(), and
          in some cases abort(), exit(), and longjmp().

          The C++ draft does say the following in 1.8 para 10:
          "When the processing of the abstract machine is interrupted by
           receipt of a signal, the values of objects modified after
           the preceding sequence point are indeterminate during the
           execution of the signal handler, and the value of any object
           not of volatile sig_atomic_t that is modified by the handler
           becomes undefined."

          This seems less restrictive than the C standard, which allows
          undefined behavior if a signal handler "refers to any object
          of static storage duration other than by assigning a value to
          a static storage duration variable of type volatile
          sig_atomic_t".

        [Erwin Unruh:]
          1.8 para 10 should be deleted.  It severely restricts
          optimizers.  We all think that in the following code

            int a,b;
            a = 7;
            b = 5;
            a = 9;

          the first assignment is optimized away.  1.8 para says a
          compiler must put the assignment down because a signal handler
          might refer to a.  I think this is an unacceptable situation
          with regard to C.

        [Erwin Unruh's proposed resolution:]
          A function registered as a signal handler may only do what it
```

is entitled to do in the C standard. A function which uses
(even potentially) a language or library feature not in C will
cause undefined behaviour.
[Note: This also covers very minor additions!]

[Example:

```
 inline void f(){}            // inline is no C
 void g(int) { if (0) f(); } // g uses a non-C feature

 signal( SIGINT, &g );       // undefined behaviour
```

Although f is never called, activating a SIGINT causes
undefined behaviour. Note that using exception handling or
RTTI would most probably cause problems on some machines.
]

The result of this discussion should go into another
paragraph in section [lib.support.runtime] 18.7.

Resolution:

Para 9 was modified to say:
"When the processing of the abstract machine is interrupted by
receipt of a signal, the values of objects with type other than
volatile sig_atomic_t are unspecified, and the value of any object
not of volatile sig_atomic_t that is modified by the handler
becomes undefined."

Requestor: Greg Colvin/Erwin Unruh
Owner: Steve Adamczyk (Sequence Points/Execution Model)
Emails:
Papers:

. . . .

==

Chapter 2 - Lexical Conventions

Work Group: Core
Issue Number: 744
Title: Is the description of trigraph processing wrong?
Section: 2.3[lex.trigraph]
Status: closed

Description:

2.3 para 4 says:
"Trigraph replacement is done left to right, so that when two
sequences which could represent trigraphs overlap, only the
first sequence is replaced. [Example: The sequence "???="
becomes "?=", not "?#". The sequence "?????????" becomes
"???", not "?". -- end example]"

[Clark Nelson, edit-778:]

> A new paragraph was added after the September draft,
> specifically [lex.trigraph]/4. The paragraph seems to be
> trying to clarify some aspects of trigraph processing.
>
> Unfortunately, the entire paragraph seems to be based on a
> false premise; to wit, that ??? is a trigraph which is
> replaced by a single ?. However, ??? is not listed as a
> trigraph sequence in the trigraph table, and according to
> paragraph 3, there are no other trigraphs. If ??? were
> a trigraph for ?, then paragraph 4 would be meaningful and,
> arguably, necessary clarification. However, if (as I believe)
> ??? is not a trigraph of any sort, then the new paragraph 4
> is actually meaningless and/or just plain wrong, and should be
> deleted.
>

> As a possibly related issue, in the C standard, the statements
> of paragraph 3 are normative. Should the note-brackets around
> that paragraph be removed from the working paper? If they were,
> the confusion about ??? might have been a little less likely.

Resolution:
Do as Clark suggests:
Paragraph 4 should be deleted and paragraph 3 should be made normative.
Requestor: Clark Nelson
Owner: Tom Plum (Lexical Conventions)
Emails:
Papers:
. . . .
Work Group: Core
Issue Number: 849
Title: Which names are reserved to implementations?
Section: 2.10 [lex.name]
Status: closed
Description:
Regarding names that are reserved for C++ implementations, Sections 2.10 and 17.3.3.1.2 both say that identifiers containing a double underscore (__) or beginning with an underscore and an upper-case letter are reserved for use by C++ implementations and standard libraries.

Section 17.3.3.1.2 also says the following:
--Each name that begins with an underscore is reserved to the implementation for use as a name with file scope or within the namespace std in the ordinary name space.

This is missing from 2.10. I assume the wording in 17.3.3.1.2 takes precedence?

2.10 should be changed to just reference 17.3.3.1.2.
Resolution:
Requestor:
Owner: Josee Lajoie (Lexical Conventions)
Emails:
Papers:
. . . .
===

Chapter 3 - Basic Concepts

Work Group: Core
Issue Number: 789
Title: When is a name used in a default argument considered "used"?
Section: 3.2[basic.def.odr]
Status: closed
Description:
Resolution:
[N1065 issue 3.32]
The working paper should explicitly state that an entity which appears to be "used" in a default argument is actually used only if the default argument itself is used.
Requestor: Bill Gibbons
Owner: Josee Lajoie (ODR)
Emails:
Papers:
. . . .
Work Group: Core
Issue Number: 664
Title: When does the reevaluation rule for class scope name lookup require a diagnostic?
Section: 3.3.6 [basic.scope.class]

```
Status:         closed
Description:
        3.3.6 para 1 says:
        2) The name N used in a class S shall refer to the same declaration
           when re-evaluated in its context and in the completed scope of S.
           No diagnostic is required for a violation of this rule.
        3) If reordering member declarations in a class yields an alternate
           valid program under (1) and (2), the program's behavior is
           ill-formed, no diagnostic is required.

        In the presence of rule 3) it is not clear why rule 2) is needed.
        The following example should be added following rule 2) to
        illustrate that rule 2) applies when a name is used in a declaration
        and then redeclared by the same declaration.

        typedef int I; //1

        class D {
            typedef I I; //2
        };
Resolution:
        The example above should be added to the WP, following rule 2) of
        3.3.6.
Requestor:      Steve Adamczyk
Owner:          Josee Lajoie (Name Lookup)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   869
Title:          Is a class name inserted in its own class scope
                considered a member name for the purpose of name look up?
Section:        3.4 [basic.lookup]
Status:         closed
Description:
        class A { };

        class X {
          class A { };
          class Y : ::A {
            A a; // base class A or X::A?
          };
        };

       The answer to this is almost clear.
       Members of base class members are found before names declared in
       containing classes (3.4.1p7), and the class name is inserted into
       the class (9p2), so I would say that the reference to A must be
       the base class.

       What is not clear is whether the insertion of the class name is
       considered to be a "member" for the purpose of 3.4.1p7.  I think
       it's intended to be, but the terminology is not consistent,
       probably because the concept of "membership" as applying to other
       than data members and member functions evolved over time.
Resolution:
Requestor:      Mike Miller
Owner:          Josee Lajoie (Name Lookup)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   686
Title:          Where is a function name looked up if an argument type is
```

```
                 introduced with a using-declaration?
Section:         3.4.2 [basic.lookup.koenig]
Status:          closed
Description:
        basic.lookup.koenig says:

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

        For each argument type T in the function call, there may be a set of
        zero or more associated namespaces to be considered; such namespaces
        are determined in the following way:
        [...]
        - If T is a class type, its associated namespaces are the namespaces
          in which the class and its direct and indirect base classes are
          defined.
        [...]
        Typedef names used to specify the types do not contribute to this
        set.

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

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

        namespace N2 {
                using N1::T;

                void f(T);
        };

        void foo() {
                N2::T t;

                f(t);           // which f?
        }
Resolution:
        The function called is N1::f.
        The sentence in 3.4.2 paragraph 2:
          "Typedef names used to specify the types do not contribute to this
           set."
        should be augmented to say that:
          "Typedef names and using-declarations used to specify the types do
           not contribute to this set."
Requestor:       Andrew Koenig
Owner:           Josee Lajoie (Name Lookup)
Emails:          core-7041
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:      Core
Issue Number:    790
Title:           What is the associated namespace if the argument has function
                 type?
Section:         3.4.2 [basic.lookup.koenig]
Status:          closed
Description:
        3.4.2[basic.lookup.koenig] para 2:
        "For each argument type T in the function call, there is a set of
         zero or more associated namespaces to be considered. The set of
```

namespaces is determined entirely by the types of the arguments."

The list does not cover arguments of function types.
An argument can have function type if the parameter has type reference to function.

Resolution:
3.4.2[basic.lookup.koenig] para 2, fifth bullet
change:
"If T is a pointer to function type, ..."
to:
"If T is a function type, ..."

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

. . . .

Work Group: Core
Issue Number: 791
Title: Does a function declaration need to be visible at the point of the call for a function call to be well-formed?
Section: 3.4.2 [basic.lookup.koenig]
Status: resolved
Description:
There should be an example to illustrate that a function name does not have to be known at the point of the call for the function call to be well-formed. i.e. parsing must not assume for:

```
name()
```

that 'name' is visible in the scope of the call for this expression to be interpreted as a function call.

```
namespace NS {
    class T{ };
    void f(T);
}
NS::T parm;
int main() {
    f(parm); //ok, calls NS::f
}
```

Resolution:
Add the suggested example.

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

. . . .

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

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

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

3.4.5 [basic.lookup.classref] says that in an unqualified name, the name after the ~ is looked up in the current context and in the class of p. But it doesn't say anything special about the qualified case. This implies that it is looked up in the scope of X only. If this is true, it seems to me that is a problem because it doesn't work when X is a typedef, as in:

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

        typedef A AB;

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

This suggests that the name after ~ should always be looked up in the current context, even for the qualified name case.

The look up for a destructor name for a class type should follow the look up of a pseudo-destructor-name (3.4.3).

Resolution:

Replace 3.4.3 [basic.lookup.qual] paragraph 5, before the example, with:

"If a pseudo-destructor-name (5.2.4) contains a nested-name-specifier, the type-names are looked up as types in the scope designated by the nested-name-specifier."

(this covers the case of the pseudo-destructor-name)

and add:

"In a qualified-id of the form:

::opt nested-name-specifier ~class-name

where the nested-name-specifier designates a namespace scope, and in a qualified-id of the form:

::opt nested-name-specifier class-name::~class-name

the class-names are looked up as types in the scope designated by the nested-name-specifier."

and clarify in 3.4.3.1[class.qual] that the qualified name look up for class members described in this subclause does not apply to the look up of a destructor name.

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

. . . .

Work Group: Core
Issue Number: 792
Title: What are the rules used to determine whether expressions involving nontype template parameters are equivalent?
Section: 3.5 [basic.link]
Status: closed
Description:

[N1053 issue 6.46]
There must be rules for determining when two template declarations/ definitions refer to the same template. For template type parameters this is obvious, but when nontype parameters are used the equivalence may involve unevaluated expressions. There must be some way to determine if two such expressions are equivalent.
The approach recommended in N1053 should be adopted.

Resolution:

3.5 should refer to the template chapter to describe when two partial specializations are valid.

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

```
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   851
Title:          Must a diagnostic be issued if main is called in a program?
Section:        3.6.1 [basic.start.main]
Status:         closed
Description:
        3.6.1 para 3 says:
        "The function main shall not be called from within a program."

        Does 'call' mean function call in the program source or does it
        refer to call during the execution of the program?  The "shall
        not" phrase can mean either that a diagnostic is required or
        that violation results in undefined behaviour depending on which
        one of these options the term 'call' refers to.

        1.3 [intro.compliance] para 5 says:
        "--Whenever this International Standard places a requirement
           on the execution of a program (that is, the values of data
           that are used as part of program execution) and the data
           encountered during execution do not meet that requirement,
           the behavior of the program is undefined and this
           International Standard places no requirements at all on the
           behavior of the program."
Proposed Resolution:
        A diagnostic is required.  "call" refers to a source code
        construct.
        Maybe the sentence should be rewritten as follows to make the
        requirement explicit:
        "A program shall not contain a call to the function main."
Resolution:
Requestor:      Steve Clamage/Fergus Henderson
Owner:          Josee Lajoie (Object Model)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   746
Title:          What is the order of initialization of a class static data
                member?
Section:        3.6.2[basic.start.init]
Status:         closed
Description:
        > On comp.std.c++, jlilley@empathy.com (John Lilley) writes:
        > The order of construction is determined by the placement of
        > the *definitions* of the static members, not the
        > declarations within the containing class.  Within a single
        > translation unit (source file), the static members are
        > constructed in the order of definition (DWP s3.6.2.1 ).

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

        As far as I can tell, the current wording of the draft leaves
        the order of initialization of static members unspecified.
Resolution:
        The wording in 3.6.2 para 1 should be changed to say instead:
        "Objects defined in namespace scope..."
Requestor:      Fergus Henderson
Owner:          Josee Lajoie (Object Model)
Emails:
```

Papers:
. . . .
Work Group: Core
Issue Number: 747
Title: The term "static initialization" needs to be defined
Section: 3.6.2[basic.start.init]
Status: closed
Description:

para 2 says:
"An implementation is permitted to perform the initialization of an object of namespace scope with static storage duration as a static initialization..."

The term 'static initialization' and 'dynamic initialization' need to be defined.

Resolution:

'static initialization' designates both zero-initialization and initialization with constant expressions.
'dynamic initialization' designates initializations that are not static initializations.

Requestor:
Owner: Josee Lajoie (Object Model)
Emails:
Papers:
. . . .
Work Group: Core
Issue Number: 852
Title: Should the destruction of array objects be inter-leaved with calls to the functions registered with atexit?
Section: 3.6.3 [basic.start.term]
Status: closed
Description:

What is the defined order of atexit-registered function calls in the following program:

```
C f() { atexit(&func1); }
C g() { atexit(&func2); }
C x[] = { f(), g() };
```

3.6.3 para 3 says:
"If a function is registered with atexit (see <cstdlib>, _lib.support.start.term_) then following the call to exit, any objects with static storage duration initialized prior to the registration of that function will not be destroyed until the registered function is called from the termination process and has completed. For an object with static storage duration constructed after a function is registered with atexit, then following the call to exit, the registered function is not called until the execution of the object's destructor has completed."

The current draft (3.6.3) indicates that, upon termination, atexit will call registered functions in the example above in the following order:

```
Destructor for x[1]
func2
Destructor for x[0]
func1
```

This result is inconsistent with the behaviour of the following slightly different program:

```
C f() { static C local1; }
```

```
  C g() { static C local2; }
  C x[] = { f(), g() };
```

The last sentence in 3.6.3 paragraph 1 says:

"For an object of array or class type, all subobjects of that object are destroyed before any local object with static storage duration initialized during the construction of the subobjects is destroyed."

This mandates that destructor be called in the following order:

```
  Destructor for x[1]
  Destructor for x[0]
  Destructor for local2
  Destructor for local1
```

Should the ordering for these two programs be consistent? Shouldn't the first program call functions registered with atexit in a the following order?

```
  Destructor for x[1]
  Destructor for x[0]
  func2
  func1
```

Resolution:
3.6.3 para 3 should say:
"If atexit is called during the construction of an object, the complete object to which it belongs shall be destroyed before the registered function is called."

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

. . . .

Work Group: Core
Issue Number: 772
Title: Wording needs to acknowledge there is no such thing as a const reference
Section: 3.9.3[basic.type.qualifier]
Status: closed
Description:
3.9.3/3 says:
"Each non-function, non-static, non-mutable member of a const-qualified class object is const-qualified, ..."

This is clearly wrong, since there is no such thing as a const-qualified reference (as opposed to a reference to const-qualified type.)

"non-reference" should be added to the list in 3.9.3/3.

7.1.1/8 says:
"The mutable specifier can be applied only to names of class data members (9.2) and cannot be applied to names declared const or static."

References are implicitly const, because a reference may not be changed to refer to another object after initialization.

The omission of "reference" in the restrictions in 7.1.1 appears to be an almost-editorial oversight.

Resolution:

```
        Clarify the WP as Bill suggests.
Requestor:      Bill Gibbons
Owner:          Steve Adamczyk (Types)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
================================================================================
 Chapter 4 - Standard Conversions
----------------------------------
Work Group:     Core
Issue Number:   885
Title:          Can array rvalues decay to pointers?
Section:        4.2 [conv.array]
Status:         closed
Description:
        Somewhere in the Kona edits, 4.2 [conv.array] paragraph 1
        changed from saying "An lvalue or rvalue of array type ...  can
        be converted to an rvalue of type pointer ..." to saying "An
        lvalue of array type ..." etc.

        96-0178/N0996 has "lvalue or rvalue", and 96-0219R1/N1037 (the
        CD2 version with change bars) has "lvalue".

        The consequence of this change is that examples like the
        following are currently ill-formed:

            struct A {
              int arr[5];
            };
            A f();
            void g() {
              f().arr[3] = 1;
            }

        This sort of example was discussed by the committee and we
        agreed it should be valid.

        The status quo makes C++ more like C.  However, it is different
        from what the ARM says and what the committee decided.
Resolution:
        4.2 para 1 now says:
        "An lvalue or rvalue of array type..."
Requestor:
Owner:          Steve Adamczyk (Type Conversions)
Emails:
Papers
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   773
Title:          When is the conversion array of const char to pointer to
                char applied on a string literal?
Section:        4.2 [conv.array]
Status:         closed
Description:
        Is the following legal?

          char* pc = "abc" + 1;

        When the string "abc" is converted from an array of const char
        to a pointer, before the '+ 1' is applied, which conversion
        takes place, the one that yields 'const char*' or the one that
        yields 'char *'?  How is it decided which array-to-pointer
        conversion is applied?
```

Of course there is more than just the + operator that can cause
this question to come up. For example,

```
("abc")
&*"abc"
```

Also, when a throw expression is a string literal, will

```
catch (char *) { }
```

catch it?

Resolution:
At the Nashua meeting, it was decided that the deprecated standard conversion from string to char* is only applied when there is an explicit target type of type char*.

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

. . . .

Work Group: Core
Issue Number: 793
Title: Is it "null pointer constant" or "null-pointer constant"?
Section: 4.10 [conv.ptr]
Status: closed
Description:
Resolution:
It is "null pointer constant".
18.1 para 4 needs to be modified.
Requestor: ANSI CD2 Public Comment 28
Owner: Steve Adamczyk (Type Conversions)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 854
Title: Must a null pointer constant be an rvalue of integer type of value 0?
Section: 4.10 [conv.ptr]
Status: closed
Description:
4.10 para 1:
"An integral constant expression (5.19) rvalue of integer type that evaluates to zero (called a null pointer constant) can be converted to a pointer type."

Is this supposed to be a definition for the "null pointer constant"? It doesn't really say that. If an A is a D, it does not mean that other things can't also be Ds. I don't find any other definition of "null pointer constant".

Could an implementation define NULL to be a zero value of a magic internal compiler type that was compatible with all pointer types but not with integral types? In that case, given

```
void f(int);
void f(char*);
```

the expression f(NULL) would call f(char*), but with a usual implementation would call f(int). In addition, usual implementations would allow

```
int i = 2 + NULL;
```

but the hypothetical implementation would flag it as an error.

```
Proposed Resolution:
        The sentence in 4.10 is intended to define the term "null
        pointer constant".

        The first two phrases of 4.10 para 1 could be reversed to show
        the intent better:
        "A null pointer constant, which is an integral constant
         expression (5.19) rvalue of integer type that evaluates to zero,
         can be converted to a pointer type."

        According to the definition of NULL in chapter 18, NULL must be
        a null pointer constant.
Resolution:
Requestor:      Steve Clamage
Owner:          Steve Adamczyk (Type Conversions)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
==============================================================================
 Chapter 5 - Expressions
-------------------------
Work Group:     Core
Issue Number:   855
Title:          ::name is not a qualified-id
Section:        5.1 [expr.prim]
Status:         closed
Description:
        The term "qualified-id" is sometimes used in the WP to
        designate a name solely prefixed by the :: operator.
        However, the grammar does not allow a qualified-id to be
        preceded by a leading ::. This should be clarified.

        For example, 3.4.4 para 1 says:
        "The class-name or enum-name in the elaborated-type-specifier
         may either be a simple identifier or be a qualified-id."
        The above does not allow:
           class ::B ....
        to refer to a global class name.
Resolution:
Requestor:
Owner:          Josee Lajoie (Name Look Up)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   794
Title:          Are recursive calls to main() allowed?
Section:        5.2.2[expr.call]
Status:         closed
Description:
        para 9 says:
        "Recursive calls are permitted."

        To match what 3.6.1 says regarding main(), this sentence should
        "Recursive calls are permitted, except to the function named
         main (3.6.1, [basic.start.main])."
Resolution:
        Add the suggested wording.
Requestor:      ANSI CD2 Public Comment 36
Owner:          Steve Adamczyk (Expressions)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
```

Issue Number: 795
Title: Should a pseudo-destructor call allow the object expression to have a different cv-qualification from the type-name naming the destructor?
Section: 5.2.4[expr.pseudo]
Status: closed
Description:

5.2.4[expr.pseudo] para 2 says:
"The left hand side of the dot operator shall be of scalar type. The left hand side of the arrow operator shall be of pointer to scalar type. This scalar type is the object type. The type designated by the pseudo-destructor-name shall be the same as the object type."

```
const int* pci;
typedef int I;
pci->~I(); //ill-formed
```

Should a pseudo-destructor call allow the object expression to have a different cv-qualification from the type-name naming the destructor?

Resolution:

Yes, the pseudo-destructor call should allow the object expression to have a different cv-qualification from the type-name naming the destructor.

The last sentence quoted above should say:
"The cv-unqualified versions of the object type and of the type designated by the pseudo-destructor-name shall be the same type."

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

. . . .

Work Group: Core
Issue Number: 774
Title: Should the WP say that converting from void* to original pointer type yields a pointer value equal to original pointer?
Section: 5.2.9[expr.static.cast]
Status: closed
Description:

[Steve Clamage:]

The C standard says explicitly that any data pointer can be converted to void* without loss of information, and that you can convert the void* back to the original type and the result will compare equal to the original pointer.

I don't find the second part of that statement for static_cast. I think we need that guarantee, so that we know for any type T:

```
T*    t1 = ...;
void* p  = t1;
assert( static_cast<T*>(p) == t1 ); // cannot fail
```

[Josee:]

5.2.9 paragraph 6 says the following:
"The inverse of any standard conversion sequence (_conv_), other than the lvalue-to-rvalue (_conv.lval_), array-to-pointer (_conv.array_), function-to-pointer (_conv.func_), and boolean (_conv.bool_) conversions, can be performed explicitly using static_cast subject to the restriction that the explicit conversion does not cast away constness (_expr.const.cast_)"

```
          A conversion from a data pointer to a void* is a standard
          conversion so the wording above allows the conversion from a
          void* to a data pointer.

          Should additional wording be added to say that the result
          will compare equal to the original pointer?
Resolution:
        Make it clear that static_cast of pointer to object type to void*
        and back again gives the original pointer value.
Requestor:      Steve Clamage
Owner:          Steve Adamczyk (Type Conversions)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   775
Title:          Is a conversion between a pointer to a struct and a
                pointer to the first member of the struct a static_cast?
Section:        5.2.9[expr.static.cast]
Status:         closed
Description:
        From comp.std.c++:
        In article 1@jake.esu.edu, jpotter@falcon.lhup.edu
        (John E. Potter) writes:
        >Steve Clamage (Stephen.Clamage@Eng.Sun.COM) wrote:
        >: Second counter-example, much stronger:
        >:  struct S { int i; ... };
        >:  S s;
        >:  int* ip = static_cast<int*>(&s); // convert struct* to int*
        >:  *ip = 2;
        >: The rules of C and C++ state explicitly that '&s' can be
        >: converted to a pointer to its first element, and therefore
        >: modifying 's' via 'ip' is completely valid.
        >
        > Yes, 9.2/17 assures that &s suitably cast to int* must work.
        >
        > But 5.2.9 [expr.static.cast] does not list pointer to POD
        > conversion to pointer to first member as one of the valid
        > conversions.

        Should the conversion in 9.2/17 be a static_cast or a
        reinterpret_cast?

        In the C standard, the section on casts does not explicitly
        mention that the conversion between a pointer to struct and a
        pointer to the first element of the struct is a valid
        conversion.
Resolution:
        At the Nashua meeting, the core WG decided that the static_cast
        from a pointer to struct to a pointer to the first member of the
        struct should remain invalid. A reinterpret_cast should be used
        instead.
        Question:
        Wording is probably needed in the reinterpret_cast subclause to
        indicate that such a reinterpret_cast is well-defined?
Requestor:      Steve Clamage
Owner:          Steve Adamczyk (Type Conversions)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   858
Title:          Can an expression of any type be cast to its own type
```

```
                using a reinterpret_cast?
Section:        5.2.10 [expr.reinterpret.cast]
Status:         closed
Description:
        This complements issue 796.

        5.2.10 para 2 says:
        "Any expression may be cast to its own type using a
         reinterpret_cast operator."

        There are two problematic cases with this scenario:
        (1)  Array types.
        It's a little weird to be able to cast an lvalue array to its
        own (array) type.
        (2)  Class types.  Maybe it's okay to cast a class expression to
        its own type, but what are the semantics?  Is a copy made?  If
        so, presumably the copy constructor is not called. (?)

        Both could be resolved by saying that the reinterpret_cast does
        nothing in that case, i.e., it's like a set of parentheses, but
        even there, one would have to be careful to indicate whether
        the expression is forced to an rvalue.
Proposed Resolution:
        All things considered, it seems it would be better to make a
        change here like the one recommended for const_cast (Issue 796).
Resolution:
Requestor:      Steve Adamczyk
Owner:          Steve Adamczyk (Type Conversions)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   796
Title:          Can a const_cast cast _any_ type to its own type?
Section:        5.2.11 [expr.const.cast]
Status:         closed
Description:
        para 2 says:
        "Any expression may be cast to its own type using a const_cast
         operator."

        Can this be applied to types not normally valid as const_cast
        operands?
Resolution:
        It should be made clear that casting an operand to its own type
        using a const_cast is ok as long as the type is valid for an
        operand of a const_cast. (i.e. pointer, pointer-to-member or
        reference).

        [Josee: Shouldn't this restriction also be applied to
        reinterpret_cast? Para 2 of 5.2.10 also allows any operand to be
        cast to its own type using a reinterpret_cast.]
Requestor:
Owner:          Steve Adamczyk (Type Conversions)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   860
Title:          Is ptr->~T() a call to a destructor?
Section:        5.3.1 [expr.unary.op]
Status:         closed
Description:
        5.3.1 para 9:
```

"There is an ambiguity in the unary-expression ~X(), where X
 is a class-name. The ambiguity is resolved in favor of treating
 the ~ as a unary complement rather than treating ~T as
 referring to a destructor."

This seems to contradicts 12.4 [class.dtor] para 12:

```
struct B {
        virtual ~B() { }
};
struct D : B {
        ~D() { }
};

D D_object;
typedef B B_alias;
B* B_ptr = &D_object;
...
B_ptr->~B();          // calls D's destructor ??complement op??
B_ptr->~B_alias();    // calls D's destructor ??complement op??
...
```

Should 5.3.1 para 9 say that it only applies if the unary
operator is not part of a postfix expression?

Resolution:
What follows a . or -> is not a unary expression, it is an
id-expression.

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

. . . .

Work Group: Core
Issue Number: 886
Title: What arguments are passed to placement operator delete?
Section: 5.3.4 [expr.new]
Status: closed
Description:
In [expr.new] (5.3.4), para 19 says:
"A declaration of placement operator delete matches the
 declaration of a placement operator new when it has the same
 number of parameters and ... all parameter type except the
 first are identical."

but para 20 says:
"If placement operator delete is called, it is passed the same
 arguments as were passed to placement operator new."

Same arguments, even for the first parameter?

Proposed Resolution:
Para 20 should say same _additional arguments_.

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

. . . .

Work Group: Core
Issue Number: 669
Title: semantics for new and delete expressions should be
separated from the requirements for operator new and
delete
Section: 5.3.4 [expr.new], 5.3.5 [expr.delete]
Status: closed
Description:

Erwin Unruh wrote a paper (96-0011/N0829) that suggested that the semantics for the new expression and the delete expression be reworked so that they would only describe which operator new (or operator delete) they call. The restrictions on the behavior of the allocation and deallocation functions called should be moved to the library section.

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

5.3.4 New

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

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

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

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

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

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

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

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

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

5.3.5 Delete

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

[Note: this means that the syntax of the delete-expression must
match the type of the object allocated by new, not the syntax of
the new-expression.]"

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

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

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

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

. . . .

Work Group: Core
Issue Number: 797
Title: Is initialization performed if the nothrow operator new
returns a null pointer value?
Section: 5.3.4 [expr.new]
Status: closed
Description:

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

If nothrow operator new is called and returns NULL, initialization
should not be done (and the deallocation function should not be
called).

Resolution:

At the Nashua meeting, the committee members seemed to favor this
resolution:

"If the library nothrow operator new (or its user-defined
replacement) returns a null pointer value, no initialization is
done."

Requestor: ANSI CD2 Public Comment 28
Owner: Josee Lajoie (Memory Model)
Emails:
Papers:

. . . .

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

[Fergus Henderson in core-7251:]

> The following example in a note in 12.4/13 is not strictly
> conforming C++ according to the rules defined elsewhere in the
> draft. I think it should be changed.
>
> "13[Note: explicit calls of destructors are rarely needed. One
> use of such calls is for objects placed at specific addresses
> using a new- expression with the placement option. Such use

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

[Mike Miller in core-7257:]

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

```
        >    to allow the correct invocation of destructors).
        > 3) Make char and unsigned char a special case, like they are
        >    in many other ways, such that allocating an array of char
        >    or unsigned char is guaranteed to have an "array allocation
        >    overhead" of zero.
        > I guess I don't have a strong preference among the three,
        > although 2 and 3 seem a bit more straightforward and
        > correspond more to the rest of the language.
        >
        > This is obviously not a make-or-break issue; people will
        > continue to write "new char[xx]" and it will continue to work,
        > whether we bless it or not.  But it's not hard to change the
        > WP to allow it, and it would bring us a little closer to
        > reality to recognize this particular practice.
Resolution:
        The WP should be changed to allow "new char[xx]" to get a
        suitably-aligned buffer space for other objects:

        5.3.4 paragraph 9
        replace:
          "When the allocation function is called, the first argument shall
           be the amount of space requested (which shall be no less than the
           size of the object being created and which may be greater than the
           size of the object being created only if the object is an array)."
        with:
          "When the allocation function is called, the first argument shall
           be the amount of space requested.  If the object being created is
           not an array, the size requested by the new expression to
           operator new shall be the size of the object.  If the object is an
           array, the size requested by the new expression to operator new
           may be larger than the size of the object.  For arrays of char
           and unsigned char, the difference between the result of the new
           expression and the address returned by the allocation function
           shall be an integral multiple of the most stringent alignment
           requirement (3.9) of any object type whose size is no greater than
           the size of the array being created. [Note: since allocation
           functions are assumed to return pointers to storage that is
           appropriately aligned for objects of any type, this constraint on
           array allocation overhead permits the common idiom of allocating
           character arrays into which objects of other types will later be
           placed. ]

        Also the first line of the example above should be deleted. The
        library placement new is not replaceable.
Requestor:      Fergus Henderson
Owner:          Josee Lajoie (Memory Model)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   719
Title:          Is unsigned arithmetic modulo 2~N for multiplication as well?
Section:        5.6 [expr.mul]
Status:         closed
Description:
        5.6/3, Binary * operator

        According to 3.9.1/3, unsigned arithmetic is always modulo 2^N.
        For addition and subtraction this is easy to remember, but for
        multiplication the rule should probably be repeated here since
        it is less obvious.
Resolution:
Requestor:      Bill Gibbons
Owner:          Steve Adamczyk (Expressions)
```

```
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   798
Title:          What are the semantics of pointer +/- enum?
Section:        5.7 [expr.add]
Status:         closed
Description:
Resolution:
        Para 1 should make it clear that, in pointer +/- enum, the enum
        is treated as an integral type that is the underlying type of
        the enum.
Requestor:
Owner:          Josee Lajoie (Memory Model)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   721
Title:          Comparisons of pointer to class members need fine tuning
Section:        5.9 [expr.rel]
Status:         closed
Description:
        5.9/2 says:
          "If two pointers point to nonstatic data members of the same
           object, the pointer to the later declared member compares
           greater provided the two members are not separated by an
           access-specifier label (11.1) and provided their class is not
           a union."

        The "point to" provision probably should also cover "point
        within".
Resolution:
        The WP should be clarified to also cover "point within".
Requestor:      Bill Gibbons
Owner:          Josee Lajoie (Memory Model)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   799
Title:          An example illustrating comparisons of pointers to different
                types and different cv-qualifications is needed
Section:        5.9 [expr.rel]
Status:         closed
Description:
        Para 2 says:
        "Pointer conversions and qualification conversions are performed on
         pointer operands to bring them to their composite pointer type. ...
         Otherwise, the composite pointer type is a pointer type similar
         (4.4) to the type of one of the operands, with cv-qualification
         signature (4.4) that is the union of the cv-qualification
         signatures of the operand types."

        This could be clarified by adding an example.
Resolution:
        In Nashua, the core WG agreed, an example would be helpful.
Requestor:      ANSI CD2 Public Comment 23
Owner:          Josee Lajoie (Memory Model)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
```

Issue Number: 861
Title: Should the WP say that &x == &y is false if x not same object as y?
Section: 5.10 [expr.eq]
Status: closed
Description:

The relational operators (5.9) produce unspecified results when comparing addresses of unrelated objects. I'm using "unrelated" to mean that neither is a subobject of the other, neither is a subobject of the same object, and they are not both part of the same array. I also am referring to the built-in address-of operator, not an user-defined operator. Prototypical example:

```
void f() {
   int x, y;
   bool b = (&x <= &y); // unspecified result
   ...
```

So far, so good.

Section 5.10 says the equality operators have the same rules as the relationals, but goes on to provide some cases when addresses must compare equal. Conspicuously absent is any statement about equality of addresses of unrelated objects.

```
bool b = (&x == &y); // also unspecified!
```

I thought I remembered a guarantee that (&x!=&y) in early drafts of the C standard, but the current C standard does not make the guarantee. (So far as I can tell. Fergus Henderson thinks the C standard is open to interpretation on that point but I don't see why. It is irrelevant in any case, since the C++ standard could tighten the requirement without causing any problems. Surely there is no program that depends on x and y having addresses that compare equal!)

It seems like a peculiar omission, since we generally expect the address of an object to determine its identity. In particular, I think much of STL relies on the proposition:

(&x==&y) if and only if x and y are the same object

Was the "only if" part of the proposition deliberately left out, and if so, can someone explain why?

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

. . . .

Work Group: Core
Issue Number: 722
Title: The definition of address constant expression needs fine tuning
Section: 5.19 [expr.const]
Status: closed
Description:

5.19/4 address constant expressions

This needs work. For example, the phrase "The subscription operator ... can be used" does not describe how it may be used; presumably the subscript must be an integral constant expression.

The same goes for 5.19/5.

Resolution:

```
        The following text should be added to paragraph 4 and 5:
        "If the subscript operator is used, one of its operands shall be an
         integral constant expression."
Requestor:      Bill Gibbons
Owner:          Josee Lajoie (Initialization)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
=============================================================================
Chapter 7 - Declarations
--------------------------
Work Group:     Core
Issue Number:   800
Title:          Mistake in description of when an incomplete class can be
                used
Section:        7.1.1[dcl.stc]
Status:         closed
Description:
        7.1.1 para 8 says:
        "The name of a declared but undefined class [...] cannot be used
         before the class has been defined."

        This should say: "can be used in ways that do not require a complete
        class type (3.2)".
Resolution:
        Do as suggested above.
Requestor:
Owner:          Josee Lajoie (Object Model)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   862
Title:          A local name declared const does not have internal
                linkage
Section:        7.1.5.1 [dcl.type.cv]
Status:         closed
Description:
        7.1.1 para 6 says:
        "A name declared in a namespace scope without a storage-class-
         specifier has external linkage unless it has internal linkage
         because of a previous  declaration  and  provided  it  is not
         declared const.  Objects declared const and not explicitly
         declared extern have internal  linkage."

        but 7.1.5.1 para 2 misses the `namespace scope' part (i.e., it
        forgets about objects with no linkage, I think):

        "An object declared with a const-qualified type  has  internal
         linkage unless it is explicitly declared extern or unless it
         was previously declared to have external  linkage.[...]"
Proposed Resolution:
        7.1.5.1 should say:
        "An object declared in a namespace scope ...".
Resolution:
Requestor:      David Vandevoorde
Owner:          Josee Lajoie (Linkage)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   683
Title:          What is the underlying type of an enumeration type if the
                value of an enumerator uses the value of a previous
```

enumerator?
Section: 7.2 [dcl.enum]
Status: closed
Description:

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

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

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

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

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

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

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

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

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

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

The answer may seem obvious, but consider:

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

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

There seem to be several possible solutions to this problem:

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

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

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

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

Bill Gibbons' preference is (1).

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

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

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

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

Resolution:
        At the Nashua meeting, the core WG decided that option (1) should be
        implemented. i.e. When an enumerator constant is used before the
        closing "}" of its enumeration, it should have the type of the
        initializing expression.

Requestor: Bill Gibbons
Owner: Steve Adamczyk (Type Conversions)
Emails: core-6989
Papers:

. . . .

Work Group: Core
Issue Number: 672
Title: using-declarations and base class assignment operators
Section: 7.3.3 [namespace.udecl]
Status: closed

Description:
        7.3.3 should indicate what happens if a using-declaration refers to
        a base class assignment operator and the type of this assignment
        operator corresponds to the type of the derived class copy assignment
        operator.

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

Resolution:
        At the Nashua meeting, members of the core WG wanted the implicit
        copy assignment operator for class B to still be generated.

        Add at the end of 7.3.3[namespace.udecl] paragraph 4:
        "If an assignment operator brought from a base class into a derived
        class scope has the signature of a copy assignment operator for the
        derived class (12.8), the using-declaration will not by itself
        suppress the implicit declaration of the derived class
        copy-assignment operator, and if the implicitly-declared operator
        has the same parameter type as an assignment operator brought in by
        a using-declaration, that assignment operator from the base class
        will be hidden or overridden by the implicitly-declared operator, as
        described below."

        Add in 12.8 paragraph 10, after the first sentence:
        "A using-declaration (7.3.3) that brings in from a base class an
        assignment operator with one of the parameter types of a copy
        assignment operator is not considered an explicit declaration of a
        copy assignment operator, and if the base class assignment operator
        has the same parameter type as the implicitly-declared copy
        assignment operator, the operator from the using-declaration will be
        hidden by the implicitly-declared operator."

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

Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 863
Title: Can the name introduced by a using-declaration be the same as the name of an entity already declared in that scope?
Section: 7.3.3 [namespace.udecl]
Status: closed
Description:

7.3.3/1 says:
"A name specified in a using-declaration in a class or namespace scope shall not already be a member of that scope."

7.3.3/10 says:
"If the set of declarations and using-declarations for a single name are given in a declarative region,
-- they shall all refer to the same entity, or all refer to functions; or
-- exactly one declaration shall declare a class name or enumeration name and other declarations shall all refer to the same entity or all refer to functions; in this case the class name or enumeration name is hidden."

7.3.3 para 1 should probably be changed to reflect what 7.3.3 para 10 says.

. . . .

[Bill Gibbons also mentions]:
There is a note at the end of 7.3.3/13:

[Note: two using-declarations may introduce functions with the same name and the same parameter types. A call to such a function is ill-formed unless name look up can unambiguously select the function to be called (because the function name is qualified by its class name, for example).]

The note in 7.3.3/12 says the same thing about namespace and block scope.

Why must the ambiguity be resolved by name lookup, and not by overload resolution? For example:

```
namespace A {
    void f(int);
    void f(long);
}
namespace B {
    void f(long);
    void f(double);
}
namespace C {
    using A::f;
    using B::f;
    void g() {
        f(123);  // ill-formed ?
    }
}
```

As written, the WP makes this ill-formed because there are two different functions "f(long)" at the point of the call. Of course overload resolution would not be ambiguous.

Resolution:

Requestor: Herb Sutter
Owner: Josee Lajoie (Name Lookup)
Emails:
Papers:
. . . .
Work Group: Core
Issue Number: 801
Title: Clarification of the interaction of partial specializations and using-declarations
Section: 7.3.3 [namespace.udecl]
Status: closed
Description:
[N1053 issue 6.58]
Resolution:
Using declarations only affect the visibility of declarations occurring before the using declaration itself; they do not affect the visibility of subsequent declarations with the same name. However, partial specializations of class templates are found by looking up the primary class template and then considering all partial specializations of that template. So if a using declaration names a class template, subsequent partial specializations are effectively visible because the primary template is visible. The working paper should make this clear, and should include an example.
Resolution:
Requestor: John Spicer
Owner: Josee Lajoie (Name Look Up)
Emails:
Papers:
. . . .
Work Group: Core
Issue Number: 802
Title: Clarification of conversion template instance names and using-declarations
Section: 7.3.3 [namespace.udecl]
Status: closed
Description:
[N1053 issue 8.11]
Resolution:
It should be made clear that a using-declaration (in a derived class) may not refer to an instance of a conversion function member template (in a base class).
Resolution:
Requestor: John Spicer
Owner: Josee Lajoie (Name Look Up)
Emails:
Papers:
. . . .
Work Group: Core
Issue Number: 729
Title: Must extern "C" functions declared in a namespace and a global extern "C" function have different signatures and return types?
Section: 7.5 [dcl.link]
Status: closed
Description:
3.5[basic.link] para 10 says:
"After all adjustments of types [...], the types specified by all declarations of a name in a given namespace shall be identical [...]."

Because this says "of a name in a given namespace", it does not cover the following properly:

```
extern "C" int f(int);
```

```
        namespace NS {
            extern "C" void f(int); // ill-formed? undefined behavior?
        }

        because the "C" function is declared in difference namespaces.
Resolution:
        Amend 3.5[basic.link]p10 to read:
        "After all adjustments of types (during which typedefs
         (_dcl.typedef_) are replaced by their definitions), the types
         specified by all declarations referring to a given object or
         function shall be identical, except that declarations for an array
         object can specify array types that differ by the presence or
         absence of a major array bound (_dcl.array_). A violation of this
         rule on type identity does not require a diagnostic."

        Amend the first two sentences of 7.5[dcl.link]p6 to read:
        "At most one object or function with a particular name can have C
         linkage. Two declarations for an object or function with C language
         linkage with the same object or function name (ignoring the
         namespace names that qualify it) that appear in different namespace
         scopes refer to the same entity."
Requestor:
Owner:          Josee Lajoie (extern "C")
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   749
Title:          Can a declaration specify both a storage class and a
                linkage specification?
Section:        7.5[dcl.link]
Status:         closed
Description:
        What is the meaning of:

           extern "C" static void f();

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

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

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

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

              extern "C" int i; // declaration
          "

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

              extern static void f();
```

```
          and that Mike's solution (1) is the correct one.
Resolution:
        Add to 7.5[dcl.link] at the end of paragraph 7:
        "A linkage-specification directly containing a single declaration
         shall not specify a storage class. [For example:
              extern "C" static void f(); // error
         -- end example]
        "
Requestor:      Mike Anderson
Owner:          Josee Lajoie (extern "C")
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   750
Title:          To which declarator in a member function declaration does
                the extern "C" specifier apply?
Section:        7.5[dcl.link]
Status:         closed
Description:
        [Mike Miller in core-7322]:
        > What is the meaning of 7.5p4, "A non-C++ language linkage is
        > ignored ... for the function type of class member function
        > declarators" with respect to parameters of member functions?
        > For instance,
        >
        >         extern "C" {
        >                 struct S {
        >                         void f(void(*)());
        >                 };
        >         }
        >
        > Does S::f take a "C" function or a "C++" function?  The
        > example in the text deals with related issues but not this
        > specific one, and the normative text could be read either way,
        > depending on whether you understand "function type of class
        > member function declarators" in a shallow or deep sense.

        [Mike Anderson in core-7323:]
          I believe it was intended to be understood in a shallow sense
          (and that S::f takes a "C" function).  The words were crafted
          to make the rule apply only to certain function types (namely,
          those of member function declarators) and not to any other
          function types such as the types of function parameters.

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

        [Mike Miller in core-7325:]
          Assuming that we do intend the "shallow" interpretation, I
          think the normative words there are wrong; the type of S::f is
          different ("function taking pointer to C function...") from
          what it would be if it were not inside extern C ("function
          taking pointer to C++ function..."), i.e., the non-C++ linkage
          is *not* ignored in determining the function type.  IMHO, it
          should be rewritten to read something like, "The language
          linkage of member names and member function types is C++,
          regardless of the linkage specification in which the class may
          be defined." (An example is also a good idea.)
Resolution:
        It should be made clear that the sentence quoted in 7.5 para 4
        applies to the member function in a shallow sense.
```

```
        The sentence should be rewritten to read something like, "The
        language linkage of member names and member function types is C++,
        regardless of the linkage specification in which the class may be
        defined."
        (An example is also needed.)
Requestor:      Mike Miller
Owner:          Josee Lajoie (extern "C")
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
==============================================================================
 Chapter 8 - Declarators
-------------------------
Work Group:     Core
Issue Number:   776
Title:          Name look up in default argument expressions
Section:        8.3.6 [dcl.fct.default]
Status:         closed
Description:
        para 5 says:
          "The names in the expression are bound, and the semantic
           constraints are checked, at the point of declaration."

        At the point of declaration of what? the function or the
        parameter?

        In this example, to which 'f' does the default argument refers?
        ::f or N::f?

        typedef int (*PF)();
        int f(PF);
        namespace N {
          int f(PF p = &f);
        }
Resolution:
Requestor:
Owner:          Steve Adamczyk (Default Arguments)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   865
Title:          What is the potential scope of a function parameter?
Section:        8.4 [dcl.fct.def]
Status:         closed
Description:
        Subclause 3.3.2 paragraph 2 reads:
        "The potential scope of a function parameter name in a function
         definition (_dcl.fct.def_) begins at its point of declaration.
         If the function has a function try-block the potential scope
         of a parameter ends at the end of the last associated handler,
         else it ends at the end of the outermost block of the function
         definition.  A parameter name shall not be redeclared in the
         outermost block of the function definition nor in the
         outermost block of any handler associated with a function
         try-block."

        But subclause 8.4 paragraph 2 reads:
        "The parameters are in the scope of the outermost block of the
         function-body."

        I presume the latter sentence should simply be removed.  The
        following shows why it makes a difference.
```

```
           const int n = 1;
           void f(int n,
                  int m = n); // which n?
```

Resolution:
Requestor: Neal Gafter
Owner: Josee Lajoie (Name Lookup)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 751
Title: Should { } be allowed around an initializer that is a string?
Section: 8.5[dcl.init]
Status: closed
Description:

The current WP disallows:

```
   const char a[3] = {"asdf"};
```

However, this is allowed in C.

8.5 paragraph 13 says:
"If T is a scalar type, then ...

```
    T x = { a };
```

is equivalent to

```
    T x = a;
```

"

An array is not a scalar type.

If the committee decides to leave things the way they are, this difference between C and C++ should be listed in appendix C.

Resolution:

Redundant { } should be allowed around string initializers.

In 8.5.2[dcl.init.string] paragraph 1, after each occurence of "can be initialized by a string literal" insert "optionally enclosed in braces".

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

. . . .

Work Group: Core
Issue Number: 867
Title: copy constructors do not have parameters of derived class type
Section: 8.5 [dcl.init]
Status: closed
Description:

Editorial Issue:
In the definition of copy-initialization in section 8.5 para 14, footnote 89 says:
"Because the type of the temporary is the same as the type of the object being initialized, or is a derived class thereof, this direct-initialization, if well-formed, will use a copy constructor (_class.copy_) to copy the temporary."

The term "copy constructor" is not used correctly here.
Direct initialization considers not only the copy constructor but all constructors such that a constructor that accepts a derived class type would be preferred in this situation:

```
  struct D;
  struct B {
```

```
    B(const B&);
    B(const D&);
  };
  struct D : B { };

  struct X {
    operator D();
  };

  X x;
  B b = x;
```

Isn't the temporary created by this copy-initialization of type D (i.e. "the type returned by the call of the user-defined conversion function")? Shouldn't B(const D&) be selected? B(const D&) is not a copy constructor.

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

. . . .

Work Group: Core
Issue Number: 868
Title: description of aggregate initialization should refer to default initialization
Section: 8.5.1 [dcl.init.aggr]
Status: closed
Description:

8.5.1 para 7 says that "each member not explicitly initialized shall be initialized with a value of the form T()".

This should instead say that the member should be default-initialized. This matters when the type T is an array, because you can't write T() for an array type T.

If this change is made, paragraph 8 (about leaving a reference uninitialized) can be made a note, because default-initialization of a reference is ill-formed ([dcl.init] para 5).

12.6.1 para 2 should also talk about default initialization.

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

. . . .

Work Group: Core
Issue Number: 804
Title: Can a reference bind directly to what a function call returns if the function returns a reference?
Section: 8.5.3[dcl.init.ref]
Status: closed
Description:

```
struct A {};
struct B {
        operator A&();
};
B f();
A &r1 = f(); // Should this be allowed?
```

The WP does not allow the previous statement.
However, many compilers give no error on the above statement.

```
        const A &r2 = f(); // should a copy always be made?

        This last case is valid according to the WP, but the implementation
        is required to copy the result of the conversion function to a
        temporary, and bind the reference to that. This extra copy is also
        not existing practice.
Resolution:
        The WP should allow the first initialization.
        The WP should not require that a temporary be created for the second
        statement.
        See Steve Adamczyk's paper 97-0012/N1050 for proposed wording.
Requestor:      Steve Adamczyk
Owner:          Steve Adamczyk (Type Conversions)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
==============================================================================
 Chapter 9 - Classes
---------------------
Work Group:     Core
Issue Number:   805
Title:          Can a zero-size class contain static members, member
                functions and nested types?
Section:        9[class]
Status:         closed
Description:
        9[class] para 3 says:
        "A class with an empty sequence of members and base class objects is
         an empty class.  Complete objects and member subobjects of an empty
         class type shall have nonzero size.1)
         1) That is, a base class subobject of an empty class type may have
            zero size.
        "

        struct SS {
                typedef int I;
                static int C;
                void f();
        };

        SS does not have an empty sequence of members. Why can't it have a
        zero-size?
Resolution:
        The definition of empty class is not needed.
        9 para 3, the first two sentences and the footnote should be replaced
        with:
          "Complete objects and member subobjects of class type shall have
           nonzero size.
           Footnote: base class subobjects are not so constrained."
Requestor:      Nathan Myers
Owner:          Josee Lajoie (Object Model)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   870
Title:          Is an error required if a static data member is used
                and not defined?
Section:        9.4.2 [class.static.data]
Status:         closed
Description:
        9.4.2 para 2 says:
        "A definition shall be provided for the static data member if
         it is used (3.2) in a program."
```

```
        9.4.2 para 5 says:
        "There shall be exactly one definition of a static data member
         that is used in a program; no diagnostic is required;

        Para 2 does not say: "no diagnostic required".

        The duplication and difference between these two sentences is
        a bad thing. The sentence in 9.4.2 para 2 should be removed.
Resolution:
Requestor:
Owner:          Josee Lajoie (Object Model)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   505
Title:          Must anonymous unions declared in unnamed namespaces also be
                declared static?
Section:        9.5 [class.union] Unions
Status:         closed
Description:
        9.5p3 says:
        "Anonymous unions declared at namespace scope shall be declared
         static."
        Must anonymous unions declared in unnamed namespaces also be declared
        static?
        If the use of static is deprecated, this doesn't make much sense.
Resolution:
        An alternative should be to declare the anonymous unions as members
        of an unnamed namespace. When the static keyword is removed,
        it will not be possible to declare anonymous unions in namespace
        scope unless the anonymous unions are declared in an unnamed
        namespace.

        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:          Josee Lajoie (Linkage)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   871
Title:          Can a class with a constructor but with no default
                constructor be a member of a union?
Section:        9.5 [class.union]
Status:         closed
Description:
        9.5[class.union]:
        "An object of a class with a non-trivial default constructor
         (_class.ctor_), a non-trivial copy constructor (_class.copy_), a
         non-trivial destructor (_class.dtor_), or a non- trivial copy
         assignment operator (_over.ass_, _class.copy_) cannot be a
         member of a union, nor can an array of such objects."

        This should say, "An object with a non-trivial constructor".
        i.e.
          class C {
            C(int);
          };

          Objects of type C cannot be members of a union.
```

```
Resolution:
Requestor:
Owner:          Josee Lajoie (Object Model)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
==============================================================================
 Chapter 10 - Derived classes
------------------------------
Work Group:     Core
Issue Number:   624
Title:          class with direct and indirect class of the same type: how
                can the base class members be referred to?
Sections:       10.1 [class.mi] Multiple base classes
Status:         closed
Description:
        para 3 says:
        "[Note: a class can be an indirect base class more than once and can
         be a direct and indirect base class.]"
        The WP should describe how base class members can be referred to,
        how conversion to the base class type is performed, how
        initialization of these base class subobjects takes place.
Resolution:
        A note will be added to the WP to clarify the restrictions on
        accessing members of the direct base class.

        Add after the 2nd sentence of paragraph 3:
        "There are limited things that can be done with such a class.
         The non-static data members and member functions of the direct
         base class cannot be referred to in the scope of the derived
         class. However, static members, enumerations and types can be
         unambiguously referred to."
Requestor:
Owner:          Josee Lajoie (Object Model)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
==============================================================================
 Chapter 11 - Member Access Control
------------------------------------
Work Group:     Core
Issue Number:   806
Title:          11 para 1  does not cover all members that can refer to the
                private and protected members of a class
Section:        11[access]
Status:         closed
Description:
        11[access] para 1 only lists a subset of the members that can refer
        to the private and protected members of a class.

        "A member of a class can be
         --private; that is, its name can be used only by member functions,
           static  data  members, and friends of the class in which it is
           declared.
         --protected; that is, its name can be used only by member functions,
           static data members, and friends of the class in which it is
           declared and by member functions, static data members, and friends
           of classes derived from this class (see _class.protected_).
        "

        The description should be made more general.
Resolution:
        The first two bullets should be replaced with:
        "-- private; that is, its name can be used only by members and
```

```
            friends of the class in which it is declared.
         -- protected; that is, its name can be used only by members and
            friends of the class in which it is declared and by members
            and friends of classes derived from this class (see 11.5)."
Requestor:
Owner:          Steve Adamczyk (Access)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   752
Title:          When accessing a base class member, the qualification is not
                ignored
Section:        11.5[class.protected]
Status:         closed
Description:
        11.2 para 4 says:
        "The access to a member is affected by the class in which the
         member is named.  This naming class is the class in which the
         member name was looked up and found.  [Note: this class can be
         explicit, e.g., when a qualified-id is used, or implicit, e.g.,
         when a class member access operator (_expr.ref_) is used
         (including cases where an implicit this->" is added.  If both a
         class member access operator and a qualified-id are used to
         name the member (as in p->T::m), the class naming the member is
         the class named by the nested-name-specifier of the
         qualified-id (that is, T).  ]"

         This is contradictory to the example in 11.5 para 1:

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

            class D1 : public B {
            };

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

         According to 11.2 para 4, the qualification is not ignored.
Resolution:
Requestor:
Owner:          Steve Adamczyk (Access)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   807
Title:          Can local classes within member functions refer to the
                private members of the member function's class?
Section:        11.8[class.access.nest]
Status:         closed
Description:
        11.8 para 1 says:
```

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

Is the following example well-formed?

```
class A {
public:
        void B();
private:
        enum X { X1, X2, X3 };
};
void A::B() {
        struct Z { X x; int i; };
}
```

Can local classes within member functions refer to the private members of the member function's class?

Resolution:
Clarify that a local class has the same access to a containing class as does the containing function (i.e. the local class is not a nested class).

Requestor: ANSI CD2 Public Comment 16
Owner: Steve Adamczyk (Access)
Emails:
Papers:

. . . .

==

Chapter 12 - Special Member functions

Work Group: Core
Issue Number: 808
Title: During the construction of a const object, what happens if the object is modified, and a pointer to const type assumes that the object remains unchanged?
Section: 12.1[class.ctor]
Status: closed
Description:
During the construction of a const/volatile object, the constructor and, functions called by the constructor, can modify the object under construction. Does this mean that the implementation cannot use optimization techniques (like assume that a const object does not change during the execution of a function) for functions called by constructors?

```
struct C;
void no_opt(C*);

struct C {
    int c;
    C() : c(0) { no_opt(this); }
};

const C cobj;

void no_opt(C *cptr)
{
    int i = cobj.c * 100;
    cptr->c = 1; // must the implementation assume that
                 // cobj is modified by this assignment?
    cout << cobj.c * 100 << '\n';
}
```

Resolution:
Requestor: Randy Meyers
Owner: Josee Lajoie (Object Model)
Emails:

Papers:

. . . .

Work Group: Core
Issue Number: 777
Title: Should it be mentionned in 12.2 that the exception object has a lifetime longer than the full-expression?
Section: 12.2[class.temporary]
Status: closed
Description:

12.2 paragraph 4 says:
"There are two contexts in which temporaries are destroyed at a different point than the end of the full-expression."

Should this also discuss the exception object created when an exception is thrown? The exception object created in the run-time may be perceived as a temporary but its lifetime is longer than the full-expression.

Resolution:

It should be made clear that the exception object is not a temporary affected by the rules in this subclause.

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

. . . .

Work Group: Core
Issue Number: 874
Title: Clarify lifetime of temporary example
Section: 12.2 [class.temporary]
Status: closed
Description:

12.2 paragraph 5 example:

```
"  class C {
       // ...
       public:
          C();
          C(int);
          friend const C& operator+(const C&, const C&); // problem
          ~C();
       };
   C obj1;
   const C& cr = C(16)+C(23);
   C obj2;
```

the expression C(16)+C(23) creates three temporaries. A first temporary T1 to hold the result of the expression C(16), a second temporary T2 to hold the result of the expression C(23), and a third temporary T3 to hold the result of the addition of these two expressions. The temporary T3 is then bound to the reference cr."

Binding the result of the expression to "C const& cr" is a nice example of a const reference to a temporary; however, the function does not return a temporary, it returns a "C const&". With the snapshot of the example given, it is very difficult (impossible?) to determine where T3 came from. If the function returns a "C" rather than a "C const&", everything makes sence.

Resolution:
Requestor: John Potter via Steve Clamage
Owner: Josee Lajoie (Object Model)
Emails:
Papers:

. . . .

Work Group: Core

```
Issue Number:   809
Title:          It should be made clear that when the destructor for a
                derived class implicitly calls the destructor for a base
                class, the virtual function mechanism is not used
Section:        12.4[class.dtor]
Status:         resolved
Description:
        12.4[class.dtor]:
        Make it clear that a derived class destructor implicitly calls a base
        class destructor such that the virtual function mechanism is never
        used.
Resolution:
        After the first sentence of paragraph 6, add the following sentence:
        "All destructors are called as if they were referenced with a
         qualified-id, i.e. ignoring any possible virtual overriding
         destructors in more-derived classes."
Requestor:      Anthony Scian
Owner:          Josee Lajoie (Object Model)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   810
Title:          When a class has a member and a base class with the same name
                what does a mem-initializer-id referring to this name
                designate, the base or the member?
Section:        12.6.2 [class.base.init]
Status:         closed
Description:
        A note should indicate that when a class has a base and a member
        with the same name, a mem-initializer-id designates the class member
        and it is not possible to refer to the base class in a
        mem-initializer-id.
Resolution:
        Add the following note after the first sentence of para 2 in
        12.6.2[class.base.init]:
        "[Note: if a class has a member with the same name as one or its
          direct or virtual base, a mem-initializer-id for a constructor of
          this class naming the member or base class and composed of a single
          identifier references the class member.  A mem-initializer-id for
          the hidden base class may be specified using a qualified name.]"
Requestor:      CD2 Public Comment 20 4)
Owner:          Josee Lajoie (Object Model)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   875
Title:          If a constructor has no ctor-initializer, but the class
                has a const member, is the constructor definition
                ill-formed?
Section:        12.6.2 [class.base.init]
Status:         closed
Description:
        The CD is clear that the following:

        struct A {
           ~A();
        };

        struct Y {
           Y() : d(0.0) {}
           A const a;
           double d;
```

```
};
```

is ill-formed because the mem-initializer-list for Y does not include an initializer for `a' (which is a const non-POD class without a user-declared default-ctor). [class.base.init]/4

However, if Y were defined as:

```
struct Y {
   Y() {}
   A const a;
};
```

then the answer is not clear: the rules for mem-initializer-lists do not apply since there is no mem-initializer-list.

Proposed Resolution:
The intention was that it be ill-formed.
In the opening sentence of [class.base.init]/4, we should add "(including the case where there is no mem-initializer-list because the constructor definition has no ctor-initializer)".

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

. . . .

Work Group: Core
Issue Number: 811
Title: Can a base class copy assignment operator that is virtual be overriden by an assignment operator declared in a derived class?
Section: 12.8[class.copy]
Status: closed
Description:

```
struct B {
        virtual B& operator=(const B&);
};
struct D : B {
        B& operator=(const B&);
};
```

If D's copy assignment operator is implicitly defined, does it call B's copy assignment operator such that the virtual function mechanism is not used:

```
B::operator=(...)
```

or such that the virtual function mechanism is used:

```
((B*)(this))->operator=(...)
```

to initialize its base class?

Resolution:
The virtual mechanism is not used.

Replace the first bullet of 12.8[class.copy], para 13, with:
"-- if the subobject is of class type, the copy assignment operator is used (as if by explicit qualification, i.e., ignoring any possible virtual overriding functions in more derived classes);"

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

. . . .

Work Group: Core
Issue Number: 876a
Title: The optimization that allows a copy of a class object to

```
                alias another object is too permissive
Section:        12.8 [class.copy]
Status:         closed
Description:
        12.8 [class.copy] Paragraph 15.
        A comment on comp.std.c++ said the following:
        "This paragraph is fundamentally flawed and should either be
         removed or substantially reworked (preferably removed).  The
         "optimisation" it describes allows the compiler to arbitrarily
         violate the basic semantic axiom that arguments passed by value
         are not modified."

        Andrew Koenig replies:
        > In c++std-core-7448, John Skaller discusses a potential
        > problem with the rule that says, in effect, that for
        > optimization purposes a compiler is allowed to assume that
        > copy constructors copy their objects and that the original and
        > the copy can be aliased if one of them is never used again.
        >
        > I think the problem can be summarized by saying that objects
        > can bind resources, and even if an object is not used, the
        > resource it binds might be.  The kind of thing that might
        > happen is
        >
        >  Thing x = /* some value */;
        >  SubThing y = x.extract_portion();
        >  Thing z = x;
        >  z.clobber_portion();
        >  // now try to fetch the value of y
        >
        > If x is never used again, the compiler is entitled to alias z
        > and x.  However, if y actually refers to part of the storage
        > that x used, clobbering z (which is an alias to x) might also
        > clobber y.
        >
        > I can think of a few ways of dealing with this problem:
        >
        >  1. Acknowledge that the problem exists, but don't solve it.
        >
        >  2. Outlaw the optimization except in very restricted
        >     circumstances.
        >
        >  3. Offer a way for class authors to say `Don't optimize'
        >
        >
        > I haven't decided whether or not I think (1) is a good idea,
        > but I don't think (2) is a good idea, unless we put a whole
        > lot of work into defining the cases.  The reason is that the
        > optimization makes a tremendous difference in fairly common
        > cases like these:
        >
        > class Point {
        >   // ...
        >   int x, y;
        >   // ...
        >
        >   Point& operator=+(Point p) {
        >     x += p.x; y += p.y; return *this;
        >   }
        >   // ...
        > };
        >
        > inline Point operator+(Point p, point q) {
        >   Point r = p;
```

```
>   r += q;
>   return r;
> }
>
> Perhaps the author should have used const Point& instead of
> just Point, but not every does.  Anyway, the optimization
> allows the compiler to rewrite the parameters of operator+ to
> avoid copying them, even if Point has an explicit copy
> constructor.  I'd hate to lose that.
>
> On the other hand, I have a simple way of allowing for (3):
> just say that if a class has an `explicit' copy constructor,
> that means that the compiler is not allowed to optimize it
> away (with the possible exception of the return value
> optimization).  I suspect that anyone who knows enough to
> define classes that play aliasing games will know enough to
> say `explicit'.

[Fergus Henderson, core-7469]
> I suspect that at the time it [allowing the aliasing] was
> considered, the committee may not have considered the
> implications of the word "unused".
>
> Just as we have "bitwise const" and "logical const", so we
> can talk about "bitwise use" and "logical use".  Which sort
> of "use" does 12.8 para 15 refer to?
>
> The optimization in question is reasonable if and only if
> the original is subsequently logically unused.  This has lead
> some people (e.g. Pete Becker) to interpret the current text
> as referring to logical use, and I suspect that many people
> voting for the resolution may have been implicitly assuming
> that use meant logical use.
>
> However, if your machine does not have a "read the
> programmer's mind" instruction, then logical use is not
> computable.  If the text is interpreted to mean logical use,
> then the paragraph becomes non-normative waffle, because no
> earthly compiler can take advantage of it.
>
> So as I see it, the status quo is that the working paper is
> ambiguous.  If "use" was intended to mean "logical use", as I
> suspect it may have been, then (due to problems that were not
> noticed at the time) the text that was voted in turns out to
> be useless, and so it should be deleted.  If "use" was
> intended to mean "bitwise use", as it generally does
> elsewhere in the WP, then the text that was voted in is
> useful, but breaks some programs that really ought to be
> legal (and again, I suspect that these problems were not
> really understood at the time it was voted in).
>
> Given that this distinction between bitwise use and logical
> use was not made clear at the time (please correct me if I'm
> wrong), and given that the problems that the bitwise use
> version causes were not made clear at the time (again, please
> correct me if I'm wrong), I think that the committee ought to
> reconsider this issue.
```

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

. . . .

==

Chapter 13 - Overloading

```
Work Group:     Core
Issue Number:   778
Title:          How does the implicit argument match the implicit parameter
                of a base class static member function?
Section:        13.3.1[over.match.funcs]
Status:         closed
Description:
        13.3.1 para 4 says the following:

        "For static member functions, the implicit object parameter is
         considered to match any object (since if the function is
         selected, the object is discarded)."

        This implies that the following:
        struct S {
           S(int) { }
           void f(int) { }
           static void f(const S&) { }
           void foo() { f(1); } // call f(1) is _not_ ambiguous
        };

        struct D : public S {
           void bar() { f(1); } // call f(1) is ambiguous
        };

        I [Josee] find this a bit surprising.
        An example above should be added to the WP.

        Or, is this behavior really intended?
        If not, the wording in 13.3.1 should say that the implicit
        object argument is not always an exact match for the implicit
        parameter, and that the conversion described in 13.3.3.1.4
        (i.e. the raking of an initialization for a reference to a base
        class type initialized with a derived class object is Conversion
        Rank) also applies to the implicit object argument of a static
        member function.
Resolution:
        At the Nashua meeting, the core WG agree that 13.3.3 should indicate
        that the "conversion sequence" on the implicit object parameter for a
        static member function is no better, no worse than other conversion
        sequences (and therefore is never the deciding factor in selecting
        one function over another).
Requestor:
Owner:          Steve Adamczyk (Type Conversions)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   812
Title:          Is the built-in operator for , & -> used if overload
                resolution is ambiguous?
Section:        13.3.1.2[over.match.oper]
Status:         closed
Description:
        13.3.1.2 para 9 says:
        "If the operator is operator , , the unary operator &, or the
         operator ->, and overload is unsuccessful, then the operator is
         assumed to be the built-in operator and interpreted according to
         clause 5".

        What does 'unsuccessful' mean?
        Is the built-in operator used if overload resolution is ambiguous?
```

```
Resolution:
        "unsuccessful" means "no viable functions are found" and does not
        include ambiguity.
Requestor:      ANSI CD2 Public Comment 13
Owner:          Steve Adamczyk (Type Conversions)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   877
Title:          13.3.1.6 isn't about binding to a temporary
Section:        13.3 [over.match]
Status:         closed
Description:
        13.3 para 2 says:
        "Overload resolution selects the function to call in seven
         distinct contexts within the language:
         ...
         --invocation of a conversion function for initialization of a
           temporary to which a reference (_dcl.init.ref_) will be
           directly bound (_over.match.ref_)."

        But 13.3.1.6 [over.match.ref] isn't about binding to a
        temporary, it's about binding to an lvalue.

        13.3.1.6 [over.match.ref] para 1 says:
        "Under the conditions specified in _dcl.init.ref_, a reference
         can be bound directly to an lvalue that is the result of
         applying a conversion function to an initializer expression."
Resolution:
Requestor:      Jason Merrill
Owner:          Steve Adamczyk (Overload Resolution)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   813
Title:          The partial ordering rules for function templates are overly
                restrictive
Section:        13.3.3 [over.match.best]
Status:         closed
Description:
        [N1065 issue 1.15]
        13.3.3 para 1:
        "-- F1 and F2 are template functions with the same signature, and
            the function template for F1 is more specialized than the
            template for F2 according to the partial ordering rules
            described in 14.5.5.2, ..."

        The partial ordering rules for function templates are overly
        restrictive: they require that two functions being compared have
        identical signatures.  This restriction could be relaxed to just
        require that the functions have identical parameter types for
        overloading purposes.
Resolution:
Requestor:      Bill Gibbons
Owner:          Bill Gibbons (Templates)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   733
Title:          Implicit conversion sequences and scalar types
Section:        13.3.3.1 [over.best.ics]
```

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

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

        Jason Merrill also notes in core-7309:

        > In this test case, I assert that under the current overloading
        > rules the second and third functions are equally good matches for
        > the argument, even though the third is "obviously" the right
        > choice.  The ics for the third a reference binding to the lvalue,
        > while the ics for the second is a reference binding to a temporary,
        > but that also has identity rank because there are no lvalue->rvalue
        > conversions for built-in types.  Perhaps there should be?
        >
        >  int f(char &);
        >  int f(const char &);
        >  int f(volatile char &);
        >  int f(const volatile char &);
        >
        >  int main()
        >  {
        >    volatile char c = 'a';
        >    f (c);
        >  }

        To which Stephen Adamczyk replies:

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

        [Josee:]
        I think a note or footnote should be added to make this clear.
        I have seen many trip over this.
Resolution:
Requestor:
Owner:          Steve Adamczyk (Type Conversions)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   779
Title:          identity conversion is preferred over lvalue-to-rvalue
                conversion
Section:        13.3.3.2[over.ics.rank]
Status:         closed
Description:
        Subclause 13.3.3.2 paragraph 3, third sub-bullet has the
```

```
        following example:

          int g(const int&);
          int g(int);
          int i;
          int k = g(i);      // ambiguous

        The call to g is not ambiguous.
        The match to g(const int&) is identity.
        The match to g(int) requires an lvalue-to-rvalue conversion.

        The first sub-bullet of paragraph 3 says that:
          "the identity conversion sequence is considered to be a
           subsequence of any non-identity conversion sequence"
        because of this rule, g(const int &) is be preferred.
Resolution:
Requestor:
Owner:          Steve Adamczyk (Type Conversions)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   682
Title:          operator ?: and operands of enumeration types
Section:        13.6 [over.built]
Status:         closed
Description:
        The type of a conditional expression choosing between two enums of
        the same type was changed in the May WP from that enum type to the
        integral type it promotes to, breaking code.  I propose changing
        paragraph 27 of 13.6 [over.built] from

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

        to

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

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

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

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

        The builtin candidates are:
            operator ?(bool, const char &, const char &)
            operator ?(bool, int, int)
Resolution:
Requestor:      Jason Merrill
Owner:          Steve Adamczyk (Type Conversions)
Emails:         core-6983, core-6987
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
```

```
Work Group:     Core
Issue Number:   734
Title:          ambiguity in "bool & ? void *& : classType&" where
                classType has an operator void*&
Section:        13.6 [over.built]
Status:         closed
Description:
        This testcase is ambiguous under the current rules:
          void *p;

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

          bool b;
          A a;

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

        The implementation of the current rules results in:
          Ambiguous overload for `bool & ? void *& : A &'
          candidates are: operator ?:(bool, void *&, void *&) <builtin>
                          operator ?:(bool, void *, void *) <builtin>
        because there is no lvalue->rvalue conversion to disambiguate
        for non-class operands.
Resolution:
Requestor:      Jason Merrill
Owner:          Steve Adamczyk (Type Conversions)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   756
Title:          most uses of built-in "?" with class operands are
                ambiguous
Section:        13.6[over.built]
Status:         closed
Description:
        The pseudo-prototype for the "?" operator in [over.built] makes
        most uses of "?" with a class operand ambiguous.

        Consider

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

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

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

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

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

        are indistinguishable.  As [over.best.ics] para 6 says, in a
        copy-initialization, "Any difference in top-level cv-qualification
        is subsumed by the initialization itself and does not constitute a
        conversion."
Resolution:
Requestor:      Steve Adamczyk
Owner:          Steve Adamczyk (Type Conversions)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
==============================================================================
 Chapter 14 - Templates
------------------------
Work Group:     Core
Issue Number:   780
Title:          The definition of 'template-declaration' is incomplete
Section:        14 [temp]
Status:         closed
Description:
        14p1 states:
        "The declaration in a template-declaration shall declare or
         define a function or a class, define a static data member of a
         class template, define a member function or a member class of a
         class template, or define a member template of a class.  ..."

        But what about...

          template <class T>
          class A {
            class B {
              static int x;
            };
          };
          template <class T>
            int A<T>::B::x = 0; // not one of allowed forms

        How can we define a static data member of a class nested within a
        class template?
Resolution:
        The list of possible forms of a template-declaration does not
        include corresponding definitions of class members where the class
        is nested within a class template, nor does it include definitions
        of member templates (whether in non-template classes, template
        classes or classes nested within one of these).
Requestor:      Neal Gafter
Owner:          Bill Gibbons (Templates)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   757
Title:          Can a template member function be overloaded?
Section:        14[temp]
Status:         closed
Description:
        14 paragraph 5 says:
        "The name of a class template shall not be declared to refer to
         any other template, class, function, object, enumeration,
         enumerator, namespace, or type in the same scope
         (_basic.scope_).  Except that a function template can be
```

overloaded either by (non-template) functions with the same
name or by other function templates with the same name
(_temp.over_), a template name declared in namespace scope
shall be unique in that namespace."

This paragraph forgets to say that (except for overloading) the
name of a function template in class scope must not be the same
as the name of any other class member.

Resolution:
The restriction that a function template name must be unique
within a namespace scope (except for overloading) should also
apply to member function templates, i.e. it should apply to class
scope as well.

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

. . . .

Work Group: Core
Issue Number: 814
Title: The semantics of the keyword "export" need to be clarified
Section: 14[temp]
Status: closed

Description:
The semantics, use and intent of the keyword "export" need to be
clarified.

Resolution:
Requestor: ANSI CD2 Public Comment 29
Owner: Bill Gibbons (Templates)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 878
Title: Can a template declaration not followed by a definition
specify export?
Section: 14 [temp]
Status: closed

Description:
[John Spicer, core 7399:]:
Can a template that is only declared (and not defined) in a
translation unit be declared "export"?

The WP says:
"A non-inline template function or static data member template
is called an exported template if its definition is preceded
by the keyword export or if it has been previously declared
using the keyword export in the same translation unit."

This does not make it clear whether an exported declaration is
ill-formed or whether the "export" is simply ignored.

[Erwin Unruh, core-7407:]
We have five possible solutions:
1: Allow export only on definitions.
2: Allow export only on entities defined in that translation
unit.
3: Allow export on declarations but without semantics if not
followed by a definition.
4: Allow export on declarations with the semantic that this will
be an exported template.
5: Require export on all declarations of an exported template.

Requestor: John Spicer
Owner: Bill Gibbons (Templates)

Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 803
Title: The restrictions on default arguments in templates are not sufficiently complete
Section: 14 [template]
Status: closed
Description:

[N1065 issue 3.35]
The restrictions (in 8.3.6 para 4 and 8.3.6 para 6) on default arguments in templates are not sufficiently complete; for example, they do not specifically mention member functions of class templates and member templates.

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

. . . .

Work Group: Core
Issue Number: 730a
Title: When are default arguments for member functions of template classes semantically checked?
Section: 8.3.6 [dcl.fct.default]
Status: closed
Description:

para 5:
"The names in the expression are bound and the semantic constraints are checked at the point of declaration."

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

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

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

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

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

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

Bjarne's Proposed Resolution:

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

Mike Miller's Proposed Resolution:

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

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

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

Resolution:
Requestor: Bjarne Stroustrup
Owner: Steve Adamczyk (Default Arguments)
Emails:
Papers:
97-0024R1/N1062R1
A Discussion of the Default Argument Instantiation by Erwin Unruh

. . . .

Work Group: Core
Issue Number: 815
Title: Does the type of a template nontype parameter of array/function type decay?
Section: 14.1[temp.parm]
Status: closed
Description:
[N1053 issue 6.54]:
"Array/function decay in template parameter lists."

The implicit "decay" of array and function types to pointer types in parameter lists should also apply to nontype template parameters.

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

. . . .

Work Group: Core
Issue Number: 816
Title: There is an ambiguity on ">" with expressions written as default arguments
Section: 14.2[temp.names]
Status: closed
Description:
The working paper has rules for handling a ">" within an expression in a template-id (14.2 para 3). A similar ambiguity occurs with

expressions written as default arguments for nontype template
parameters in the parameter list of a template. The same solution
should apply.

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

. . . .

Work Group: Core
Issue Number: 883
Title: Can "template" be used to specify that an unqualified function name refers to a template specialization?
Section: 14.2 [temp.names]
Status: closed
Description:

In the following example:

```
namespace A {
    struct B { };
    template<class T> void f(T t);
}
void g(A::B b) {
    f<3>(b);
}
```

does type-dependent (Koenig) lookup apply to the lookup of "f"?
Without the explicit "<>" template arguments, the answer is
currently yes, because the lookup of "f" (other than to
determine whether it is a type) can be deferred until after the
arguments have been parsed. But with explicit template
arguments, there is no way to parse the expression without
knowing that "f" is a template.

Proposed Resolution:

We will propose that the "template" keyword be allowed in this
context so that type-dependent lookup can be used even when
there are explicit template arguments.

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

. . . .

Work Group: Core
Issue Number: 826
Title: Does the "template" keyword apply to function and static data member templates?
Section: 14.2[temp.names]
Status: closed
Description:

[N1065 issue 1.18]
Does the "template" keyword (as applied to a dependent qualified
name) apply to function and static data member templates, or just to
class templates?

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

. . . .

Work Group: Core
Issue Number: 758
Title: Can an array name be a template argument?
Section: 14.3[temp.arg]

```
Status:         closed
Description:
        14.3[temp.arg] para 3 says:
        "A template-argument for a non-type non-reference template-parameter
         shall be ...  the address of an object or a function with external
         linkage ...  The address of an object or function shall be expressed
         as &f, plain f (for function only) ..."

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

        What was probably intended is the following:
        "The address of an object or function shall be expressed as
         '&e' except when 'e' is a function or an array in which case
         it can be expressed as 'e'."
Resolution:
        The allowed forms for a template-argument corresponding to a
        non-type non-reference template-parameter do not account for the
        above implicit conversions; i.e. the "&" prior to an array name
        or function name in these cases should be optional if the values
        decay to pointers in the absence of "&".
Requestor:
Owner:          Bill Gibbons (Templates)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   759
Title:          Initializing a template reference parameter with an
                argument of a derived class type needs to be described
Section:        14.3[temp.arg]
Status:         closed
Description:
        14.3[temp.arg], paragraph 6:

        "Standard conversions (_conv_) are applied to an expression
         used as a template-argument for a non-type template-parameter
         to bring it to the type of its corresponding
         template-parameter.
         [Example:
            struct Base { /* ... */ };
            struct Derived : Base { /* ... */ };
            template<Base& b> struct Y { /* ... */ };
            Derived d;
            Y<d> yd;   // derived to base conversion
         -- end example]
        "
        Since binding an object of a derived class type to a reference
        to a base class type is not a standard conversion anymore, this
        text needs work.
Resolution:
Requestor:
Owner:          Bill Gibbons (Templates)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   760
Title:          Is a template argument that is a private nested type
                accessible in the template instantiation context?
Section:        14.3[temp.arg]
Status:         closed
```

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

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

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

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

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

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

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

Resolution:
It may be desirable to make it more clear (perhaps with an example)
that access checking is done by name, so that if a name is accessible
then it may be used in a template-id, and in the resulting
instantiation there is no restriction on access to the corresponding
template-parameter name itself.

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

. . . .

Work Group: Core
Issue Number: 782
Title: Can a value of enumeration type be used as a template
non-type argument?
Section: 14.3 [temp.arg]
Status: closed
Description:
14.3 para 3 says:
"A template-argument for a non-type non-reference
template-parameter shall be an integral constant-expression of
integral type ..."

Values of enum types should also be allowed as non-type
template arguments. The sentence above should be changed to:

"A template-argument for a non-type non-reference
template-parameter shall be an integral constant-expression of
integral or enumeration type ..."

Resolution:
The working paper should make it clear that a constant-expression
used as a template-argument for a non-type non-reference
template-parameter may also have enumeration type.

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

. . . .

Work Group: Core
Issue Number: 879
Title: What conversions can apply to a template argument to
bring it to the type of the corresponding nontype
template parameter?
Section: 14.3 [temp.arg]
Status: closed
Description:
template <int i> class S { }; S<3.3> s;
Can the template argument for the nontype template parameter i
be a floating point constant?

14.3 para 3 says:
"A template-argument for a non-type non-reference template-
parameter shall be an constant expression of integral type,
..."

14.3 para 6 says:
"Standard conversions (4) are applied to an expression used as
a template-argument for a non-type template-parameter to
bring it to the type of its corresponding template parameter."

Proposed Resolution:
For parameters of integral or enumeration type, only the
integral promotions and integral conversions are allowed (and
not, for example, floating/integral conversions). For pointer
and reference parameters, only derived-to-base conversions and
conversion to "void*" are allowed. (If "void&" is added and
conversion to "void&" is a standard conversion, then this would
be allowed also.)

(Note that array-to-pointer and function-to-pointer conversions
would always be done under the proposed resolution to 2.1; see
2.2 also.)

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

. . . .

Work Group: Core
Issue Number: 761
Title: Can the member function of a class template be virtual?
Section: 14.5.1.1[temp.mem.func]
Status: closed
Description:
14.5.1.1 paragraph 3 says:
"A member function of a class template is implicitly a member
function template with the template-parameters of its class

template as its template-parameters."
14.5.2 paragraph 3 says:
"A member function template shall not be virtual."

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

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

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

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

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

Resolution:

The term "member function template" is not used clearly here. It is not intended to mean "member template of function type", but rather "member function of a class template which, because the enclosing class is a template, behaves somewhat like a template itself".

This distinction should be made more clear. There may be similar wording problems with respect to member templates elsewhere in the working paper.

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

. . . .

Work Group: Core
Issue Number: 817
Title: Clarification of the interaction of friend declarations and partial specializations
Section: 14.5.3[temp.friend]
Status: closed
Description:

[N1053 issue 6.50]

Resolution:

It should be made clear that friend declarations are not allowed to declare partial specializations, and that a template friend declaration specifies that all instances of that template, regardless of whether implicitly generated and regardless of whether partially or completely (explicitly) specialized, are friends of the class containing the template friend declaration.

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

. . . .

Work Group: Core
Issue Number: 818
Title: Friends classes are not well covered in 14.5.3
Section: 14.5.3[temp.friend]
Status: closed
Description:
Resolution:

Para 4:
The phrase "the corresponding member function" is incorrect; the friend might be a class. So the word "function" should be deleted.

Requestor: ANSI CD2 Public Comment 12
Owner: Bill Gibbons (Templates)
Emails:
Papers:
. . . .
Work Group: Core
Issue Number: 880
Title: When does a friend declaration refer to a global function or to a template instantiation?
Section: 14.5.3 [temp.friend]
Status: closed
Description:

Now that a global function can overload a template function, when does a friend declaration in a template class refers to the global function or when it refers to a template instantiation. For example:

```
int foo(int);
template<class T> int foo(T);

template<class T> class C1 {
  friend int foo(int);
};
template<class T> class C2 {
  friend int foo(T);
};
template<class T> class C3 {
  friend int foo<int>(int);
};
```

[John Spicer's answer:]

```
> A friend declaration in which the declarator is not qualified,
> and that does not specify an explicit template argument list
> always declares a normal (i.e., nontemplate) function.  So,
> C1 makes the previously declared foo(int) a friend.  C2 does
> too, when T is int, otherwise it declares a new normal
> (nontemplate) function.  C3 always refers to instances of the
> template foo.  C3 does not make the global foo(int) a friend,
> it makes an instance of template foo a friend.
>
> One fuzzy area is what happens if you say
>
> template <class T> void f(T);
> template <class T> struct A {
>   friend void ::f(T);
> };
>
> Does the global qualifier permit this to map onto the
> template?  Without a WP change to permit this, my answer would
> be no.
```

How and when must the template specialization syntax be used with friend declarations?

```
int foo(int);
template<class T> int foo(T);
template<> int foo<double>(double);

template<class T> class S2 {
  template<> friend int foo<T>(T);
};
```

[John Spicer's answer:]

> The "template <>" is not permitted in friend declarations.

>
> Core 3 did discuss and agree upon these issues (except that
> the issue I raised about the global qualifier was not
> discussed).

Proposed Resolution:
When explicit template arguments are provided, the friend declaration refers to the specialization. Other than that, if the enclosing scope has a function template and no non-template function, the friend declares a (hidden) new nontemplate function, exactly as if the function template declaration did not exist.

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

. . . .

Work Group: Core
Issue Number: 819
Title: Where are partial specialization allowed?
Section: 14.5.4[temp.class.spec]
Status: closed
Description:
[N1053 issue 6.49 and issue 6.53 item 3]
It is not clear whether a partial specialization must be declared in the class or namespace of which it is a member. There are cases for member templates where such a rule would prevent specialization entirely. The restrictions, if any, should be explicitly stated in the working paper.

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

. . . .

Work Group: Core
Issue Number: 820
Title: Clarification of nontype dependency rules in partial specializations
Section: 14.5.4[temp.class.spec]
Status: closed
Description:
[N1053 issue 6.51]
The restrictions in 14.5.4 [temp.class.spec] item 2 makes a large class of partial specializations ill-formed for no apparent reason. The restriction should probably be relaxed, possibly by restoring the less restrictive wording from a previous version of the working paper.

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

. . . .

Work Group: Core
Issue Number: 821
Title: The restrictions on partial specializations based on the dependency of arguments on other arguments are too severe
Section: 14.5.4[temp.class.spec]
Status: closed
Description:
Editorial Box 6:
14.5.4 para 5:
The restrictions on partial specializations based on the dependency

        of arguments on other arguments are too severe. The restrictions
        should be relaxed where possible.
Resolution:
Requestor: Editorial Box 6
Owner: Bill Gibbons (Templates)
Emails:
Papers:
. . . .
Work Group: Core
Issue Number: 881
Title: What class-key can be used in declarations of
                specializations and partial specializations?
Section: 14.5.4 [temp.class.spec] and 14.7.3 [temp.expl.spec]
Status: closed
Description:
        Is it legal to have a specialization of a class template with a
        different class-key than that with which it was declared.
        For example, the template is declared a class and the
        specialization is declared a union.

        How about partial specializations?

        I can't find any mention of template unions at all, but I
        presume that they are allowed since there is nothing disallowing
        them.
Resolution:
Requestor: Mike Ball
Owner: Bill Gibbons (Templates)
Emails:
Papers:
. . . .
Work Group: Core
Issue Number: 822
Title: Clarification of ordering rules for nontype arguments in
                partial specializations
Section: 14.5.4.2[temp.class.order]
Status: closed
Description:
        [N1053 issue 6.52]
        The partial ordering rules for class template partial specializations
        are too restrictive with respect to nontype template parameters. The
        rules should be reformulated to allow additional obviously correct
        orderings.
Resolution:
Requestor: John Spicer
Owner: Bill Gibbons (Templates)
Emails:
Papers:
. . . .
Work Group: Core
Issue Number: 823
Title: Interaction of partial ordering with default arguments and
                ellipsis parameters
Section: 14.5.4.2[temp.class.order]
Status: closed
Description:
        [N1053 issue 6.55]
        The working paper does not give clear rules for the handling of
        default arguments and ellipsis parameters when determining the
        partial ordering of function templates.
Resolution:
Requestor: John Spicer
Owner: Bill Gibbons (Templates)
Emails:

```
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   824
Title:          In which contexts should partial ordering of function
                templates be performed?
Section:        14.5.4.2[temp.class.order]
Status:         closed
Description:
        [N1053 issue 6.56]
        In addition to overload resolution, there are additional contexts in
        which partial ordering of function templates could be used to resolve
        ambiguities between function template instances with identical
        function parameters (and possibly identical template arguments) but
        generated from different partial specializations:

        * Taking the address of a template function instance

        * Matching a declaration of an instance with a particular partial
          specialization (for friend declarations, explicit specialization
          and explicit instantiation)

        * Selecting a placement delete function that matches a placement new
          operation.

        It might be useful to apply the partial ordering rules in these
        contexts.
Resolution:
Requestor:      John Spicer
Owner:          Bill Gibbons (Templates)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   825
Title:          Clarification of rules for partial specializations of member
                class templates
Section:        14.5.4.3[temp.class.spec.mfunc]
Status:         closed
Description:
        [N1053 issue 6.53 items 1 & 2]
        When a member template of a class template is partially specialized,
        the partial specializations should apply to all instances generated
        from the enclosing class template.

        When the primary template is specialized for a given instance of the
        enclosing class, none of the partial specializations of the original
        primary template should be carried over.
Resolution:
Requestor:      John Spicer
Owner:          Bill Gibbons (Templates)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   762
Title:          How can function templates be overloaded?
Section:        14.5.5.1[temp.arg]
Status:         closed
Description:
        14.5.5.1 para 4 says:
        "The signature of a function template consists of its function
         signature, its return type and its template parameter list.
         The names of the template parameters are significant only for
```

establishing the relationship between the template parameters
and the rest of the signature."

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

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

Resolution:
An example and/or text should be added to make it clear that two distinct function templates may have identical function parameter lists and that they overload, even if overload resolution alone cannot distinguish them.

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

. . . .

Work Group: Core
Issue Number: 763
Title: Partial Specialization: the transformation also affects the function return type
Section: 14.5.5.2[temp.func.order]
Status: closed
Description:
14.5.5.2 [temp.func.order] paragraph 2 says:
"The transformation used is:
-- For each type template parameter, synthesize a unique type and substitute that for each occurrence of that parameter in the function parameter list.
-- For each non-type template parameter, synthesize a unique value of the appropriate type and substitute that for each occurrence of that parameter in the function parameter list."

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

because 14.5.2 para 5 says:
"If more than one conversion template can produce the required type the partial ordering rules (14.5.5.2) are used to select the "most specialized" version that can produce the required type."

But conversion functions don't have parameters, only return types.

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

. . . .

Work Group: Core
Issue Number: 736
Title: How can/must typename be used?
Section: 14.6 [temp.res]
Status: closed
Description:
Is typename required in situations where we know only type names can be used?

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

```
        template <class T> class C {
          typename C<T> ...
        };
        ---------------------------
        What if typename is used preceding an non-dependant name?  Is
        typename ignored, or is this ill-formed?

        class A { };
        template <class T> class C {
          typename A ...
        };
```
Resolution:
Requestor:
Owner: Bill Gibbons/John Spicer (Templates)
Emails:
Papers:
. . . .
Work Group: Core
Issue Number: 764
Title: undeclared name in template definition should be an error
Section: 14.6[temp.names]
Status: closed
Description:
        The example in 14.6 paragraph 1 has the following lines:

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

        The first line is also ill-formed because a7 is not declared.
Resolution:
        In the example, the line "T::A* a7;" is ill-formed because "a7" is
        not dependent and has not been declared. The example should make
        this clear.
Requestor:
Owner: Bill Gibbons (Templates)
Emails:
Papers:
. . . .
Work Group: Core
Issue Number: 766
Title: How do template parameter names interfere with names in
                nested namespace definitions?
Section: 14.6.1[temp.local]
Status: closed
Description:
        14.6.1[temp.local] paragraph 6 says:
          "In the definition of a member of a class template that
           appears outside of the class template definition, the name
           of a member of this template hides the name of a
           template-parameter.
           [Example:
```
              template<class T> struct A {
                      struct B { /* ... */ };
                      void f();
              };

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

"

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

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

John Spicer's proposed resolution:
You should get the same result whether the function is
defined in the class (or namespace) or outside of it.
The "B" in N::f gets the template parameter B, not the
namespace member B.

Resolution:
The working paper should make it clear that although class template
members may hide template-parameter names, there is no such hiding
with namespace members since the namespace scope is entirely outside
the template declaration.

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

. . . .

Work Group: Core
Issue Number: 827
Title: C is not equivalent to C<T> when C is qualified
Section: 14.6.1 [temp.local]
Status: closed
Description:
Editorial Box 8:
Resolution:
The equivalence within the scope of a class template between the name
of a template and the corresponding template-id should not apply when
the name of the template is qualified.
Resolution:
Requestor: Editorial Box 8
Owner: Bill Gibbons (Templates)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 784
Title: The examples in 14.6.2 on dependent names need work
Section: 14.6.2 [temp.dep]
Status: closed
Description:
The examples in paragraphs 2 and 3 of 14.6.2 are still there
and are still nonsense. They need to be deleted.
Also, ANSI CD2 Public Comment 7 & 23.
Resolution:
Some of the examples in this section are in disagreement with the
textual description of dependent names and lookup rules. The
examples should be corrected or removed.

Also:
[N1065 issue 3.30]
The sentence "X<T>::a has type double." should be moved to a comment

in the example, as in:
```
    template<class T> struct X : B<T> {
        A a;   // "a" has type "double"
    };
```
Requestor: John Wilkinson
Owner: Bill Gibbons (Templates)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 828
Title: In what contexts is the use of a qualifier to look in the current template a special case not subject to the usual dependent type restrictions?
Section: 14.6.2 [temp.dep]
Status: closed
Description:

[N1065 issue 1.14]
In the following example:

```
    template<class T> struct A {
        typedef int B;
        A<T>::B b;
    };
```

is the lookup of B considered dependent?
If so, is the example ill-formed?
In what contexts is the use of a qualifier to look in the current template a special case not subject to the usual dependent type restrictions?
Under what circumstances is a base class member found using a derived class qualifier of this form?

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

. . . .

Work Group: Core
Issue Number: 829
Title: 14.6.2 para 5 should not only apply when a base class is a template parameter but also when it is a dependent type
Section: 14.6.2 [temp.dep]
Status: closed
Description:
Resolution:

Para 5:
The phrase "If a template-argument is a used as a base class..." should be changed to match the intent in para 4, e.g. "If a base class is a dependent type...".

Requestor: ANSI CD2 Public Comment 8
Owner: Bill Gibbons (Templates)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 884
Title: no diagnostics required for semantics errors in template definitions
Section: 14.6.3 [temp.nondep]
Status: closed
Description:

14.6.3 para 1 has the following example:
```
template<class T> class Z {
```

```
        public:
          void f() {
            h++; // error cannot increment function
          }
        };
```

Maybe the comment should also indicate that an implementation doesn't have to diagnose this if the template is not instantiated.

Something similar to the example in 14.6 paragraph 5 would be helpful:
// may be diagnosed even if ... is not instantiated.

Proposed Resolution:
The example in 14.6.3 should make it clear that although an implementation is allowed to diagnose this kind of error when processing the template definition, it is not required to diagnose such errors until the point of instantiation.

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

. . . .

Work Group: Core
Issue Number: 767
Title: Where should the point of instantiation of class templates be discussed?
Section: 14.6.4.1[temp.point]
Status: closed
Description:
14.6.4.1[temp.point]:
Shouldn't this subclause also discuss the point of instantiation of class templates?

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

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

Resolution:
There should be cross-references between the various paragraphs discussing points of instantiation, with respect to class, function and static data member templates.

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

. . . .

Work Group: Core
Issue Number: 830
Title: Are the rules describing the point of instantiation of a function templates too complex?
Section: 14.6.4.1[temp.point]
Status: closed
Description:
Editorial Box 11:
The rules describing the point of instantiation for function templates may be overly complex.
Consideration should be given to simplifying them.

Resolution:
Requestor: Editorial Box 11
Owner: Bill Gibbons (Templates)
Emails:

Papers:
. . . .
Work Group: Core
Issue Number: 831
Title: Should candidate functions without external linkage in other translation units render a call ill-formed?
Section: 14.6.4.2[temp.dep.candidate]
Status: closed
Description:
Editorial Box 12:
This section says that if visibility of candidate functions with external linkage in additional translations units affects the meaning of the program, the behavior is undefined. The possiblility of extending the rule to include candidate functions without external linkage should be considered.
Resolution:
Requestor: Editorial Box 12
Owner: Bill Gibbons (Templates)
Emails:
Papers:
. . . .
Work Group: Core
Issue Number: 832
Title: Difference between the rules in 14.6.5 and 3.4.2 regarding friend function name look up
Section: 14.6.5 [temp.inject]
Status: closed
Description:
14.6.5 para 2:
The example does not match the argument-dependent name lookup rules for friends stated in 3.4.2 [basic.lookup.koenig].

The rules in 3.4.2 do not match those presented to the committee when the extended argument-dependent name lookup rules were added.
Resolution:
Requestor: ANSI CD2 Public Comment 23
Owner: Bill Gibbons (Templates)
Emails:
Papers:
. . . .
Work Group: Core
Issue Number: 833
Title: The definition of "specialization" for member templates is missing
Section: 14.7 [temp.spec]
Status: closed
Description:
Editorial Box 13:
Paragraph 1:
This paragraph does not really describe the handling of member templates and of members of classes nested within class templates. The missing cases should be added.
Resolution:
Requestor: Editorial Box 13
Owner: Bill Gibbons (Templates)
Emails:
Papers:
. . . .
Work Group: Core
Issue Number: 834
Title: Does "delete ap;", where ap's type is a template specialization, cause the template to be instantiated?
Section: 14.7.1 [temp.inst]
Status: closed

Description:
Resolution:
It should be made clear that a class template is instantiated in any context where the completeness of the type might have an effect on the semantics of the program.
Requestor: ANSI CD2 Public Comment 6
Owner: Bill Gibbons (Templates)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 835
Title: Does the instantiation of a class template cause the instantiation of the class static data members?
Section: 14.7.1 [temp.inst]
Status: closed
Description:
Resolution:
The working paper should explicitly state that the implicit instantiation of a class template does not cause the implicit instantiation of the definition of a static data member, and therefore does not (by itself) cause the initialization (and associated side-effects) of static data members to occur.
Requestor: Bill Gibbons
Owner: Bill Gibbons (Templates)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 786
Title: The description of explicit instantiation does not allow the explicit instantiation of members of class templates (including member functions and static data members)
Section: 14.7.2 [temp.explicit]
Status: closed
Description:

```
template<typename T>
struct Outer {
   struct Inner { T* t_; };
};

template struct Outer<int>::Inner; // Or what?
```

[temp.explicit]/2 seems to disallow this:

```
  "The syntax for explicit instantiation is:
      explicit-instantiation:
              template declaration
   where the unqualified-id in the declaration shall be either
   a template-id or, where all template arguments can be
   deduced, a template-name.  [Note: the declaration may declare
   a qualified-id, in which case the unqualified-id of the
   qualified-id must be a template-id.]"
```

This wording in [temp.explicit] is not correct. It disallows the instantiation of members of class templates (including member functions and static data members).
Resolution:
The description should be extended to include all the members, and members of members, for which explicit instantiation is appropriate.
Requestor: Daveed Vandevoorde
Owner: Bill Gibbons (Templates)
Emails:
Papers:

. . . .

```
Work Group:     Core
Issue Number:   836
Title:          What is the point of instantiation for a specialization
                to which an explicit instantiation directive applies?
Section:        14.7.2 [temp.explicit]
Status:         closed
Description:
        Editorial Box 14:
        An explicit instantiation directive should be a point of
        instantiation for each function and static data member to which the
        directive applies.  At other points of instantiation (except
        end-of-translation-unit) for functions and static data members, the
        point of instantiation does not apply to the definition of the
        template unless the definition is needed at that point (e.g. inline
        functions, and static data members for which the the value might be
        required at compile time).
Resolution:
Requestor:      Editorial Box 14
Owner:          Bill Gibbons (Templates)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   837
Title:          When can an empty template argument list "<>" be omitted?
Section:        14.7.2 [temp.explicit] and 14.7.3 [temp.expl.spec]
Status:         closed
Description:
        The situations in which an empty template argument list "<>" may be
        omitted should be more clearly explained, particularly in the
        examples in these sections.

        Also:
        14.7.3 para 6, para 16
        The examples in these two paragraphs contradict each other. It
        appears that the last line of the example in paragraph 16 should not
        contain "<>" because the definition should not be an explicit
        specialization.
Resolution:
Requestor:      ANSI CD2 Public Comment 23 and 28
Owner:          Bill Gibbons (Templates)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   787
Title:          Make it clear that a user must provide a definition for
                an explicitly specialized template; if not, the program
                is ill-formed
Section:        14.7.3 [temp.expl.spec]
Status:         closed
Description:
        14.7 [temp.spec] says:
        "A template that has been used in a way that requires a
         specialization of its definition causes the specialization to
         be implicitly instantiated unless it has been either explicitly
         instantiated or explicitly specialized."

        14.7.3 [temp.expl.spec] paragraph 5 says:
        "If a template is explicitly specialized then that
         specialization shall be declared before the first use of that
         specialization that would cause an implicit instantiation to
         take place, in every translation unit in which such a use
         occurs."
```

14.7.3 should be made clearer that the implementation expects to find a user-supplied definition for an explicit specialized template somewhere (and give an error if the implementation doesn't find one).

Resolution:

It should be clear that when a template is explicitly specialized, the unspecialized template is not used and so there is no implicit generation for the specialization. Therefore if the specialization is used it must be defined, following the rules for requiring definitions for non-template declarations. (In particular, there are some cases where a diagnostic is required and some where no diagnostic is required.)

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

. . . .

Work Group: Core
Issue Number: 838
Title: Does an explicit instantiation directive affect the compilation model for the specified instance?
Section: 14.7.3 [temp.expl.spec]
Status: closed
Description:

[N1065 issue 1.17]
Does an explicit instantiation directive affect the compilation model for the specified instance? For example, does it imply the "inclusion" model instead of the "separation" model, even when the export keyword is used?

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

. . . .

Work Group: Core
Issue Number: 840
Title: Does the prohibition on default arguments in the definition of a specialization prohibits them in the declarations of member functions of a class specialization?
Section: 14.7.3 [temp.expl.spec]
Status: closed
Description:

[N1065 issue 3.34]
14.7.3 para 3:
"Default function arguments shall not be specified in a declaration or a definition of an explicit specialization."

Resolution:

It should be made clear that the restriction on default arguments "in" explicit specializations applies only to function template explicit specializations (including member functions and member function templates where the enclosing class is not specialized), and not to member functions of class template specializations (which are not themselves specializations).

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

. . . .

Work Group: Core
Issue Number: 841
Title: Are explicit template arguments only allowed in function calls?

Section: 14.8.1 [temp.arg.explicit]
Status: closed
Description:

[N1065 issue 1.20]
According to 14.8.1, explicit template arguments may be appended to a function template name used in a call. Surely such template arguments should be allowed in other contexts in which a function name may be used, such as when taking the address of a function.

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

. . . .

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

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

Here is the reorganization Bill Gibbons suggested in private email:

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

Resolution:

There should be cross-references between the various places where template argument deduction is done.

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

. . . .

Work Group: Core
Issue Number: 768
Title: typename keyword missing in some examples
Section: 14.8.2[temp.deduct]
Status: closed
Description:

14.8.2 paragraph 10 is an error

```
  template<int i, typename T>
    T deduce(A<T>::X x,      // T is not deduced here
             T       t,      // but T is deduced here
             B<i>::Y y);     // i is not deduced here
    A<int> a;
```

```
    B<77>  b;
    int    x = deduce<77>(a.xm, 62, y.ym);
    // T is deduced to be int, a.xm must be convertible to
    // A<int>::X
    // i is explicitly specified to be 77, y.ym must be convertible
    // to B<77>::Y
```

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

A<T>::X x above should be: typename A<T>::X x
B<i>::Y y above should be: typename B<i>::Y y

Resolution:
Add the keyword typename in the two places suggested.
Requestor:
Owner: Bill Gibbons (Templates)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 842
Title: Template argument deduction rules for template conversion functions are missing
Section: 14.8.2[temp.deduct]
Status: closed
Description:
[N1065 issue 1.16]
The working paper allows member template conversion functions, and implies that their template parameters may be deduced, but does not specify the deduction rules. These rules must be stated explicitly.
Resolution:
Requestor: Bill Gibbons
Owner: Bill Gibbons (Templates)
Emails:
Papers:

. . . .

==

Chapter 15 - Exception Handling

Work Group: Core
Issue Number: 843
Title: Are "recursive" exceptions allowed?
Section: 15[except]
Status: closed
Description:
Resolution:
Clause 15 should explicitly state that multiple exceptions may be active at the same time ("recursive" exceptions). The current wording implies this but never explicitly says that this is allowed.
Requestor: ANSI CD2 Public Comment 20
Owner: Bill Gibbons (Exception Handling)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 844
Title: Does a rethrow creates a new exception?
Section: 15.1[except.throw]
Status: closed
Description:
It is not clear whether a rethrow creates a new exception which shares the exception object with the old exception, or whether the result of the rethrow is the old exception itself. If it is the

latter, then the state of the exception should probably change from "caught" to "uncaught" as a result of the rethrow. This issue is not discussed in the working paper.

Resolution:
Requestor: ANSI CD2 Public Comment 26
Owner: Bill Gibbons (Exception Handling)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 845
Title: If a string literal is thrown, what handler can catch it?
Section: 15.1[except.throw]
Status: closed
Description:
Resolution:

The example in 15.1 para 1 needs to be updated to account for the new type of string literals. Also it might be useful to point out that the special implicit cv-qualification conversion for string literals does not apply to throw-expressions.

Requestor: ANSI CD2 Public Comment 26
Owner: Bill Gibbons (Exception Handling)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 846
Title: Where does the search for a handler starts if a handler throws an exception?
Section: 15.1[except.throw]
Status: closed
Description:
Resolution:

15.1 para 2:
The wording in this paragraph about exiting a try block should actually refer to exiting just the "try" portion of the try construct. That is, a throw from within a handler should never be caught by that handler or by a handler associated with the same try.

Requestor: ANSI CD2 Public Comment 24
Owner: Bill Gibbons (Exception Handling)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 769
Title: Are the base class dtors called if the derived dtor throws an exception?
Section: 15.2[except.dtor]
Status: closed
Description:

[Mike Ball, core-7288:]

```
#include <iostream.h>

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

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

void doit() {
  derived x;
```

```
}

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

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

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

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

[Jerry Schwarz, core-7289:]

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

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

[Bjarne, core-7290:]

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

[John Skaller, core-7294:]

I assume the base destructor IS called.

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

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

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

[Erwin Unruh, core-7297:]

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

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

        Current practice:
        [Anthony Scian, core-7299:]
          I tried the program under Watcom C++, MS VC++, and Borland C++
          with the result that all three C++ implementations destructed
          the base class.
Resolution:
        When an exception is thrown from a derived class destructor, the base
        class destructor(s) should be executed.  That is, stack unwinding due
        to the throw resumes the complete destruction of the object.  This
        should be made more clear in the working paper.
Requestor:      Mike Ball
Owner:          Bill Gibbons (Exception Handling)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   788
Title:          Is it implementation defined whether the stack is unwound
                before calling terminate in all of the 8 situations described
                in 15.5.1?
Section:        15.3[except.handle]
Status:         closed
Description:
        15.3 /9 [except.handle] states that
        "If no matching handler is found in a program, the function
         terminate() is called.  Whether or not the stack is unwound
         before calling terminate() is implementation-defined."

        It should be made clear that this implementation choice applies
        only to the "no matching handler" situation (of the eight
        situations described in 15.5.1 [except.terminate]).
Resolution:
        It should be made clear that in all other cases where terminate is
        called (other than due to failure to find a matching handler), the
        stack is not unwound.  Also, there are other cases where an
        implementation might determine, before finishing a stack unwind, that
        terminate will be called during the unwind.  The working paper should
        specify whether that portion of the unwind must actually be done.
Requestor:      Jonathan Schilling
Owner:          Bill Gibbons (Exceptions)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
==============================================================================
 Annex C - Compatibility
-------------------------
Work Group:     Core
Issue Number:   680
Title:          Annex C subclause C.1 is out of date
Section:        C.1 [diff.c]
Status:         closed
Description:
        Jonathan Schilling wrote the following:

        The introduction to Annex C (Compatibility) and subclause C.1
        (Extensions) both look like they were quickly edited from the
        base document for use in the standard, but the edit missed some
        spots and left others making no sense ("... from the dialects of
```

Classic C used up till now", "... since the 1985 version of this manual"). More attention is given to Classic C than is now necessary, and the new features list is very incomplete.

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

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

Proposed Resolution:

At the Nashua meeting, the core WG agreed that C.1 should be dropped.

Resolution:

Requestor: Jonathan Schilling

Owner: Tom Plum (C compatibility)

Emails:

compat-352

Papers:

. . . .

Work Group: Core

Issue Number: 743

Title: Some anachronisms are missing from annex C

Section: C.3 [diff.anac]

Status: closed

Description:

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

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

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

C.3 Anachronisms [diff.anac]

Extensions to the C++ language may be provided by an implementation to ease the use of C programs as C++ programs or to provide continuity from earlier C++ implementations. Note that use of such extensions is likely to have undesirable aspects. An implementation providing them should also provide a way for the user to ensure that they do not occur in a source file. A C++ implementation is not obliged to provide these features.

Resolution:

At the Hawaii meeting, the C compatibility WG decided that annex C.3 should either be removed.

Requestor: Jonathan Schilling

Owner: Tom Plum (C compatibility)

Emails:

Papers:

. . . .

Annex E - Universal-character-names

Work Group: Core

Issue Number: 891

Title: The list of hexadecimal code for CJK Unified Ideographs

```
                seems incorrect
Section:        Annex E[extendid]
Status:         closed
Description:
        The list has the following:
          fe74, 5e76-fefc, ...
        5e76 should be replace by fe76.
Resolution:
Requestor:
Owner:          Tom Plum (Annex E)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
```