Doc No: X3J16/97-0084 WG21/N1122
Date: September 30th, 1997
Project: Programming Language C++
Ref Doc:
Reply to: Josee Lajoie
(josee@vnet.ibm.com)

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

This list contains the Core WG issues that were left opened after the London meeting, as well as the issues that were posted to the Core reflector between the London meeting and the deadline for this mailing. There are exactly 50 issues on the list below.

The status of the issues below is either "active" or "editorial". The active issues are those for which the committee has not agreed on a resolution yet. The editorial issues are those for which the committee agreed on a resolution but for which the WP text needs to be modified to reflect the committee's intentions.

For reference purposes, the issues that were closed at the London meeting are listed in document 97-0086/N1124.

The issues from this list that the committee decides not to address at the November meeting will be kept as possible defect reports that the committee may decide to address after the IS has been published.

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

Lexical Conventions

Annex E:
  770: The title of Annex E needs to be made shorter

Name Look Up

3.4.1 [basic.lookup.unqual]:
  850: How does name look up proceed in the parameter list of a friend function?
3.4.3.1 [class.qual]:
  893: Lookup of conversion functions conversion-type-id and of template argument names is missing when these appear in qualified-ids
3.4.5 [basic.lookup.class.ref]:
  894: How is 'f' looked up in 'p->f<...' ?
5.3.4 [expr.new]:
  690: Clarify the lookup of operator new in a new expression
7.3.3 [namespace.udecl]:
  914: How do the keywords typename/template interact with using-declarations?
Section: 7.3.3 [namespace.udecl]

8.3 [dcl.meaning]:
  887: Can an extern declaration refer to a qualified name?

Linkage / ODR

3.2 [basic.def.odr]:
  892: ODR and string literals
7.1.2 [dcl.fct.spec]:
  745: Does &inline_function yield the same result in all the translation

```
      units?
7.5 [dcl.link]:
  864: Does extern "C" affect the linkage of function names with internal
       linkage?

Initialization/Object/Memory Model
----------------------------------
3.7.3.1 [basic.stc.dynamic.allocation]:
  895: Requirements on allocation and deallocation functions are not clear
5.3.4 [expr.new]:
  896: placement new and size of buffer required
9.5 [class.union]:
  897: an a union member be inspected through another member with the same
       "common initial sequence"?
12.2 [class.temporary]:
  901: When is a temporary bound to a reference that is a local static
       variable destroyed?
12.8 [class.copy]:
  876b:Should the optimization that allows a class object to alias another
       object also allow the case of a parameter in an inline function
       aliasing its argument?
  902: When is 'template<class T> S(T);' used to generated a copy
       constructor?


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

Sequence Points/Execution Model
-------------------------------
1.8 [intro.execution]:
  694: List of full-expressions needed

Access
------
11[access]:
  872: How do access control apply to constructors/destructors implicitly
       called for static data members?
  873: How/when is access checked in default arguments of function templates?
  898: Access to template arguments used in a function return type and in the
       nested name specifier
11.2[class.access.base]:
  888: Can a class with a private virtual base class be derived from?
  899: Clarification of access to base class members
11.8 [class.access.nest]:
  900: Can a nested class access its own class name as a qualified name if it
       is a private member of the enclosing class?

Types / Classes / Unions
------------------------
3.9.1 [basic.fundamental]:
  853: Should typeid(void-expression) be allowed?

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

Types Conversions / Function Overload Resolution
------------------------------------------------
```

```
4.8 [conv.double]:
  712: Should the result value of a floating-point conversion be
       implementation-defined?
5.2.9 [expr.static.cast]:
  857: When can temporaries created by cast expressions be eliminated?
5.2.10 [expr.reinterpret.cast]:
  859: When can a pointer to member function be used to call a virtual
       function with a covariant return type?
8.5 [dcl.init]:
  866: cv-qualifiers and type conversions
13.3.3.2 [over.ics.rank]:
  903: Is a function not viable if there exists two equally good conversion
       sequences to convert an argument to the parameter type?
13.6 [over.built]:
  889: pseudo prototypes for built-in operators and operands of
       enumeration types need fine tuning
  904: The prototypes for ?: must be fixed now that lvalue-to-rvalue was
       removed

Expressions
-----------
5 [expr]:
  748: Should we say that operator precedence is derived from the syntax?


+--------+
| Core 3 |
+--------+

RTTI
----
5.2.8 [expr.typeid]:
  856: Should the WP mention the type extended_type_info?

Templates
---------
14.1 [temp.param]:
  781: Must default template-arguments be provided only on the first
       template declaration?
14.2 [temp.names]:
  765: The syntax does not allow the keyword 'template' where the text in
       14.2 says it is allowed
14.3.3 [temp.arg.template]:
  905: How does a template template argument that is a partial specialization
       match a template template parameter?
14.5.2 [temp.mem]:
  906: Does the 'this' pointer of conversion function member templates
       participate in overload resolution?
14.5.3 [temp.friend]:
  890: Clarification of the interaction of friend declarations and use
       of explicit template arguments
14.5.4 [temp.class.spec]:
  907: How can a partial specialization be used by the definition of an
       exported template?
  908: Syntax for partial specialization missing
14.6 [temp.res]:
  882: typename is not permitted in functional cast notation
14.6.1 [temp.dep.res]:
  909: Is the unqualified name of a partial specialization implicitly
       followed by template arguments in its own class scope?
14.6.4 [temp.dep.res]:
  737: How can dependant names be used in member declarations that appear
       outside of the class template definition?
14.7.1 [temp.inst]:
```

```
  910: Which part of the class member list is instantiated when a class
       template is instantiated?
14.7.3 [temp.expl.spec]:
  839: The template compilation model rules render some explicit
       specialization declarations not visible during instantiation
14.8.1 [temp.arg.explicit]:
  911: What happens if the explicit template arguments for an overloaded
       function template only match some of the variants?
14.8.2.4 [temp.deduct.type]:
  912: Template argument deduction and pointer to member function types

Exception Handling
------------------
15.5.1[except.terminate]:
  913: What happens if a terminate() handler causes terminate() to be
       reinvoked?
15.5.2[except.unexpected]:
  847: The description of "unexpected" in 18.6.2.2 differs from 15.5.2


=============================================================================
 Chapter 1 - Introduction
--------------------------
Work Group:     Core
Issue Number:   694
Title:          List of full-expressions needed
Section:        1.8 [intro.execution]
Status:         editorial
Description:
        1.8p13: "certain contexts in C++ cause the evaluation of a
        full-expression that results from a syntactic construct other
        than expression"

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

        I didn't read the cited example (8.3.6) as thoroughly as I
        might, but I didn't see anything there that explicitly said
        "this is treated like a full-expression." Probably you could
        make the case based on combining several passages together, but
        if the other ones are like this, it would take some real
        detective work to figure it out.  If someone knows what contexts
        were intended here, even if something might be omitted, it would
        be an improvement to make it explicit, either here or in the
        various contexts.

        Steve Adamczyk:
        > I looked at the wording and I agree it could be clearer.  At
        > the least we should make normative the idea that when a
        > construct is implemented by an implicit function call, the
        > entire function call is considered a full expression.  3.2p2
        > may be useful as a list of implicit references.
Resolution:
Requestor:      Mike Miller
Owner:          Steve Adamczyk (Sequence Points)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
=============================================================================
 Chapter 2 - Lexical Conventions
---------------------------------
=============================================================================
 Chapter 3 - Basic Concepts
```

```
---------------------------
Work Group:     Core
Issue Number:   892
Title:          ODR and string literals
Section:        3.2 [basic.def.odr]
Status:         active
Description:
        class C {
        public:
                void f() { "abcd"; }
        };

        If class C is included in more than one translation unit, is the
        program well-formed? The ODR does not describe how string literals
        are equivalent.  The ODR talks about *names* referring to the same
        entity, not *tokens*:

           in each definition of D, corresponding names, looked up
           according to 3.4, shall refer to an entity defined within the
           definition of D, or shall refer to the same entity, after
           overload resolution (13.3) and after matching of partial
           template specialization (14.8.3),

        As written, the WP does not require the string literal tokens to
        refer to the same entities. Should it?

        [Josee: Possible solution:]
        Replace the 2nd bullet of paragraph 5 with the following:
          -- in each definition of D, corresponding names, looked up
             according to _basic.lookup_, shall refer to an entity defined
             within the definition of D, or shall refer to the same entity,
             after overload resolution (_over.match_) and after matching of
        |    partial template specialization (_temp.over_), with the
        |    following two exceptions:
             -- a name can refer to a const object with internal or no
                linkage if the object has the same integral or enumeration
                type in all definitions of D, and the object is initialized
                with a constant expression (_expr.const_), and the value
                (but not the address) of the object is used, and the object
                has the same value in all definitions of D
        +    -- a string literal can be used if the value (but not the
        +       address) of the string literal is used and the string literal
        +       has the same value in all definitions of D
Resolution:
Requestor:      Bill Gibbons
Owner:          Josee Lajoie (ODR)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   850
Title:          How does name look up proceed in the parameter list of a
                friend function?
Section:        3.4.1 [basic.lookup.unqual]
Status:         active
Description:
        struct A {
           typedef int AT;
           void foo(AT);
        };
        struct B {
           typedef int BT;
           friend void A::foo(AT);  // does name lookup find AT?
           friend void A::foo(BT);  // does name lookup find BT?
```

```
};
```

3.4.1 is not clear describing how the scopes are searched for the parameter list of a friend function declaration when the friend function is a member function of another class. i.e. Is the scope of B ever considered?

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

. . . .

Work Group: Core
Issue Number: 893
Title: Lookup of conversion functions conversion-type-id and of template argument names is missing when these appear in qualified-ids
Section: 3.4.3.1 [class.qual]
Status: active
Description:

```
template<class T>
void g() {
    ... &X::operator D<T>;
}
```

In which scope are D and T looked up?
In the scope of X? In the context of the entire expression? In both?

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

. . . .

Work Group: Core
Issue Number: 894
Title: How is 'f' looked up in 'p->f<...' ?
Section: 3.4.5 [basic.lookup.class.ref]
Status: active
Description:

In the following example, it is clear that the 'f' should refer to the template A::f.

```
struct A {
        template <class T> void f(T);
        void g(A* p) {
                p->f<int>(1);
        }
};
```

Similarly, the following usage is currently permitted by the working paper. The working paper specifies special rules for "p->class-name-or-namespace-name::...", and a template-id is a class-name so presumably this should work.

```
template <class T> struct f {
        int i;
};

struct A : public f<int> {
        void g(A* p) {
                p->f<int>::i = 1;
        }
};
```

In both of these examples the compiler sees 'p->f<', at which point it has to decide what to do with 'f'. It is not reasonable to attempt to scan forward and determine whether a '>::' exists that matches the 'f<'.

The question that this is leading up to, of course, is how to handle examples like the following:

```
template <class T> struct f {
        int i;
};

struct A : public f<int> {
        template <class T> void f(T);
        void g(A* p) {
                p->f<int>(1);       // which of these, if any,
                p->f<int>::i = 1;   // is permitted?
        }
};
```

[John Spicer's proposed Resolution:]

```
> If the id-expression is of the form 'p->identifier<...', the
> identifier is first looked up in the class of the object
> expression.  If the identifier is not found, it is then looked up
> in the context of the entire postfix-expression.  The program is
> ill-formed if the name, when looked up in the context of the entire
> postfix expression, does not name a class or function template.
>
> This differs from the rule for looking up 'A' in 'p->A::B', but I
> think a different rule is needed for this case to avoid breaking
> code.  For example, the addition of the global template 'f' would
> render the code ill-formed if the lookup used rules similar to the
> ones used for 'p->A::B'.
>
> template <int I> void f();
> struct A {
>        int f;
>        void g(A* p) {
>                bool b = p->f < 1;
>        }
> };
```

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

. . . .

Work Group: Core
Issue Number: 895
Title: Requirements on allocation and deallocation functions are not clear
Section: 3.7.3.1 [basic.stc.dynamic.allocation]
3.7.3.2 [basic.stc.dynamic.deallocation]
Status: editorial
Description:

It is not clear which ones of the requirements in these subclauses apply to all allocation or deallocation functions (i.e. global and class allocation or deallocation functions), which ones only apply to the global allocation or deallocation functions, and which ones only apply to the library allocation or deallocation functions. This needs to be made clearer.

Other areas needing clarification:
3.7.3.2 para 3:

"The value of the first argument supplied to a deallocation function shall be a null pointer value, or refer to storage allocated by the corresponding allocation function..."
What does corresponding allocation function mean?
The intent was to say that the deallocation function used must be for a single object if the memory was allocated with the allocation function for a single object, similarly for arrays. Corresponding does not mean same parameters.

Deciding which deallocation should be used, the restriction should be on how the storage was ultimately obtained (i.e., not automatic and not static), and _not_ on the characteristics of the functions through which the value flows.

Also, it should be made clear that memory allocated with the library (nothrow) allocation function can be deallocated by the ordinary (i.e. not nothrow) deallocation function.

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

. . . .

Work Group: Core
Issue Number: 853
Title: Should typeid(void-expression) be allowed?
Section: 3.9.1 [basic.fundamental]
Status: active
Description:

[Bill Gibbons, core-7398:]
The restriction on expressions of void type in 3.9.1/9:
"An expression of type void shall be used only as an expression statement (6.2), as an operand of a comma expression (5.18), as a second or third operand of ?: (5.16), or as the expression in a return statement (6.6.3) for a function with a return type of void."

makes this code ill-formed:

```
#include <typeinfo>
void f() { }
void g() {
    typeid(f());   // ill-formed
    typeid(void);  // OK
}
```

Should expressions of type void be allowed as operands of typeid? (Note that they are already allowed as operands of ?:, so there is a precedent for allowing them.)

[Sean Corfield, core-7404:]
Should we consider this as part of the issue to relax uses of void? This just seems to be 'yet another bug' in the handling of void (that's how I view the 'unnecessary' restrictions since they get in the way of writing templates).

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

. . . .

==

Chapter 4 - Standard Conversions

Work Group: Core

```
Issue Number:   712
Title:          Should the result value of a floating-point conversion be
                implementation-defined?
Section:        4.8 [conv.double]
Status:         active
Description:
        4.8 says for floating-point conversions:
          If the [floating-point] source value is between two adjacent
          [floating-point] destination values, the result of the
          conversion is an unspecified choice of either of those values.

        yet 2.13.3 says for floating-point literals:

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

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

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

        "Except where noted"
        Should we say that operator precedence is derived from the syntax?
        The C syntax says this in a footnote. (Footnote 35).

        Here is what the C standard says in Footnote 35:
        "The syntax specifies the precedence of operators in the evaluation
         of an expression, which is the same as the order of major
         subsections of this section, highest precedence first. Thus, for
         example, the expressions allowed as the operands of the binary +
         operator (3.3.6) shall be those expression defined in 3.3.1 through
         3.3.6. The exceptions are cast expressions (3.3.4) as operands of
         unary operators (3.3.3), and an operand contained between any of the
         following pairs of operators: grouping parentheses () (3.3.1),
         subscripting brackets [] (3.3.2.1), function-call parentheses
         (3.3.2.2), and the conditional operator ?: (3.3.15)."

        Should the C++ standard say something like this?
Resolution:
Requestor:
Owner:          Steve Adamczyk (Expressions)
Emails:
Papers:
```

```
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   856
Title:          Should the WP mention the type extended_type_info?
Section:        5.2.8 [expr.typeid]
Status:         active
Description:
        Someone asked on the reflector:
        > The extended_type_info is no longer mentioned in the draft.
        > Is there a conforming way to provide extended type information
        > now?

        Bill Gibbons answered the following:
        > The working paper should say that typeid yields an lvalue
        > referring to a type_info object >>>or an object of type derived
        > from type_info<<<.
        >
        > The name "extended_type_info" should probably still appear in
        > a note, but of course it is totally non-normative.

        [Josee: Possible solution:]
        How about putting the following at the beginning of 5.2.8 para 1.

          The result of a typeid expression is an lvalue of type const
          std::type_info (_lib.type.info_) or an lvalue of a const type
          derived from std::type_info. [Note: if a type derived from
          type_info is used, this International Standard does not place any
          requirement on the name of this type though it is recommended that
          the name extended_type_info be used.] [Note: if a type derived from
          type_info is used, the description in this subclause that refers to
          an object of type type_info must be read to refer to the object of
          the derived type instead. ]

        Bill Gibbons also notes:
          > We should also make the copy constructor and assigment operator
          > protected, not private (18.5.1).
Resolution:
Requestor:      Bill Gibbons
Owner:          Bill Gibbons (RTTI)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   857
Title:          When can temporaries created by cast expressions be
                eliminated?
Section:        5.2.9 [expr.static.cast]
Status:         active
Description:
        S s;
        (S)s; // Must this cast expression create a temporary of type S?
              // Even though s has type S already?

        A more interesting example:

        class S {
           int i;
        public:
           S foo() { i = 1; return *this; }
        };

        S s;
        (S(s)).foo(); // Does this change the value of s.i?
```

5.2.9 para 2 says that a temporary is created for S(s).
Is the implementation allowed to eliminate this temporary?

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

. . . .

Work Group: Core
Issue Number: 859
Title: When can a pointer to member function be used to call a virtual function with a covariant return type?
Section: 5.2.10 [expr.reinterpret.cast]
Status: active
Description:

5.2.10 para 10 says:
"Calling a member function through a pointer to member that represents a function type that differs from the function type specified on the member function definition results in undefined behavior, except when calling a virtual function whose function type differs from the function type of the pointer to member only as permitted by the rules for overriding virtual functions."

Does the above intend to allow the following:

```
struct X { };
struct Y: X { };

struct A {
   virtual X* f();
};
struct B : A {
   virtual Y* f();
};

X* (A::*pm)() = &A::f;
Y* (B::*pm2)();
pm2 = reinterpret_cast<Y*(B::*)()>(pm);

B b;
b.*pm2(); // is this supposed to be well formed?
```

If so, then the example should be added to the WP.

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

. . . .

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

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

Proposed Resolution:

5.3.4 [expr.new] para 9 should indicate that if the object created is of class type or if the array created is an array of classes, operator new is looked up as specified in 12.5.

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

```
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:      Core
Issue Number:    896
Title:           placement new and size of buffer required
Section:         5.3.4 [expr.new]
Status:          active
Description:
        [Matt Austern:]
        I've been reading what clause 5 and clause 18 say about array
        placement new, and I just wanted to verify that my understanding is
        correct.  Here's some sample code, just to establish notation.

          char buffer[BUFSIZ];
          A* p = new(buffer) A[N];

        My understanding:
        (1) p is a pointer to the first element of an array of A.
        (2) It is not guaranteed that p and buffer are the same address.
            For example, an implementation is permitted to store some
            bookkeeping information or padding at the beginning of the
            buffer.  The beginning of the array, then, would not correspond
            to the beginning of the buffer.
        (3) buffer is required to be properly aligned for objects of type A.
        (4) BUFSIZ must be greater than or equal to N * sizeof(A) + c, where
            c is some non-negative number.  There is no way to find out what
            c is, and there is no guarantee that it will be the same from
            call to call.  (Implication: it is impossible to use array
            placement new portably.)

        Is this more or less correct? And if so, maybe 5.3.4 should say that
        this is the case?

        [Erwin Unruh:]
        One seemingly minor point was that the amount of padding cannot be
        known. Let this become implementation-defined. Most implementations
        will have padding limited to a few words of storage. So a user can
        add say 32 bytes and read the manual to check whether that is enough.

        [Erwin Unruh core-7561:]
        The following is a safe way to allocate an array in a preallocated
        buffer, with a check whether the buffer is sufficiently big. Is an
        example such as this needed in the WP?

               static const int SAFE_MARGIN = 8;
               class watcher {};
               void* operator new[] (size_t size, watcher, void * buffer,
                                     size_t buffersize)
               {
                       if ( size > buffersize )
                       {
                               // OOps not enough memory
                               throw bad_alloc (...);
                       }
                       return buffer;
               }
               typedef something A;
               char buffer[ sizeof(A) * 5 + SAFE_MARGIN ];
               A* p = new(watcher(),buffer,sizeof(buffer)) A[5];
Resolution:
Requestor:       Matt Austern
Owner:           Josee Lajoie (Memory Model)
Emails:
Papers:
```

```
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
==============================================================================
 Chapter 6 - Statements
------------------------
==============================================================================
Chapter 7 - Declarations
--------------------------
Work Group:     Core
Issue Number:   745
Title:          Does &inline_function yield the same result in all the
                translation units?
Section:        7.1.2 [dcl.fct.spec]
Status:         active
Description:
        7.1.2 para 4 says:
        "An inline functions shall be declared in every translation unit in
         which it is used and shall have exactly the same definition in every
         case (3.2)."

        It is not clear from this statement whether taking the address
        of an inline function in different translation units must yield
        the same result.

        [Bill Gibbons notes:]
        > Given the cross-reference to the ODR, the word "same" is intended
        > to mean equivalent, not unique.
Resolution:
Requestor:      Bill Gibbons
Owner:          Josee Lajoie (ODR)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   914
Title:          How do the keywords typename/template interact with
                using-declarations?
Section:        7.3.3 [namespace.udecl]
Status:         active
Description:
        Issue 1:
        ========
        The working paper is not clear about how the typename/template
        keywords interact with using-declarations:

             template<class T> struct A {
                 typedef int X;
             };

             template<class T> void f() {
                 typename A<T>::X a;      // OK
                 using typename A<T>::X;  // OK
                 typename X b;  // ill-formed; X must be qualified
                 X c;  // is this OK?
             }

        When the rules for "typename" and the similar use of "template" were
        decided, we chose to require them at every use of the dependent name
        - that is, using them once with a name does not "declare" the name to
        be a type with regard to any subsequent use of the name.

        The way to avoid writing "typename" at every use is to declare a
        typedef; then the typedef name itself is known to be a type.

        For using-declarations, we decided that they do not introduce new
```

declarations but rather aliases for existing declarations, like symbolic links.

This makes it unclear whether the declaration "X c;" above should be well-formed, because there is no new name declared so there is no declaration with a "this is a type" attribute.

(The same problem would occur with the "template" keyword when a member template of a dependent class is used).

I think these are the main options:

(1) Continue to allow "typename" in using-declarations, and "template" (for member templates) too. Attach the "is a type" or "is a template" attribute to the placeholder name which the using-declaration "declares".

(2) Disallow "typename" and "template" in using-declarations (just as class-keys are disallowed now). Allow "typename" and "template" before unqualified names which refer to dependent qualified names through using-declarations.

(3) Document that this is broken.

Issue 2:
========
Either way, one more point needs clarification.
For the first option:

```
template<class T> struct A {
    struct X { };
};

template<class T> void g() {
    using typename A<T>::X;
    X c;    // if this is OK, then X by itself is a  type
    int X;  // is this OK?
}
```

When "g" is instantiated, the two declarations of X are compatible (7.3.3/10). But there is no way to know this when the definition of "g" is compiled. I think this case should be ill-formed under the first option. (It cannot happen under the second option.)

For the second option:

```
template<class T> struct A {
    struct X { };
};

template<class T> void g() {
    using A<T>::X;
    int X;  // is this OK?
}
```

Again, the instantiation would work but there is no way to know that in the template definition. I think this case should be ill-formed under the second option. (It would already be ill-formed under the first option.)

[John Spicer's reply:]
> The "not a new declaration" decision is more of a guiding principle

```
        > than a hard and fast rule.  For example, a name introduced in a
        > using-declaration can have different access than the original
        > declaration.
        >
        > Like symbolic links, a using-declaration can be viewed as a
        > declaration that declares an alias to another name, much like a
        > typedef.
        >
        > In my opinion, "X c;" is already well-formed.  Why would we permit
        > "typename" to be used in a using-declaration if not to permit this
        > precise usage?
        >
        > In my opinion, all that needs to be done is to clarify that the
        > "typeness or "templateness" attribute of the name referenced in the
        > using-declaration is attached to the alias created by the
        > using-declaration.  This is solution #1.
Resolution:
Requestor:      Bill Gibbons
Owner:          Josee Lajoie (Name Lookup)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   864
Title:          Does extern "C" affect the linkage of function names with
                internal linkage?
Section:        7.5 [dcl.link]
Status:         active
Description:
        7.5 para 6 says the following:
        "At most one of a set of overloaded functions with a particular
         name can have C linkage."

        Does this apply to static functions as well?
        For example, is the following well-formed?

        extern "C" {
          static void f(int) {}
          static void f(float) {}
        };

        Can a function with internal linkage "have C linkage" at all
        (assuming that phrase means "has extern "C" linkage"), for how
        can a function be extern "C" if it's not extern?

        The function *type* can have extern "C" linkage -- but I think that's
        independent of the linkage of the function *name*.  It should be
        perfectly reasonable to say, in the example above, that extern "C"
        applies only to the types of f(int) and f(float), not to the function
        names, and that the rule in 7.5 para 6 doesn't apply.

        Mike's proposed resolution:
        The extern "C" linkage specification applies only to the type
        of functions with internal linkage, and therefore some of the
        rules that have to do with name overloading don't apply.
Resolution:
Requestor:      Mike Anderson
Owner:          Josee Lajoie (Linkage)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
===============================================================================
 Chapter 8 - Declarators
------------------------
```

```
Work Group:     Core
Issue Number:   887
Title:          Can an extern declaration refer to a qualified name?
Section:        8.3 [dcl.meaning]
Status:         active
Description:
        8.3 para 1 says:
        "A declarator-id shall not be qualified except for the definition of
         a member function (_class.mfct_) or static data member
         (_class.static_) or nested class (_class.nest_) outside of its
         class, the definition or explicit instantiation of a function,
         variable or class member of a namespace outside of its namespace,
         the definition of a previously declared explicit specialization
         outside of its namespace, or the declaration of a friend function
         that is a member of another class or namespace (_class.friend_)."

        This does not allow the following. Should id be allowed?

           namespace X {
             void f();
             void g() {
                extern void X::f(); // should this be allowed?
             }
           }
Resolution:
Requestor:
Owner:          Josee Lajoie (Name Look Up)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   730b
Title:          When are default arguments for member functions of template
                classes semantically checked?
Section:        8.3.6 [dcl.fct.default]
Status:         editorial
Description:
        The following bit of text in the WP does not take into account the
        resolution adopted in London regarding default arguments of function
        templates.
        para 5:
        "The names in the expression are bound and the semantic constraints
         are checked at the point of declaration."
Proposed Resolution:
        I would like to add to the text above to say that these rules do not
        apply to default arguments in template functions and refer to 14.7.1
        where the rules for default arguments are.  How about adding the
        following in the note at the end of paragraph 5?:

        "Name look up and checking of semantic constraints for default
         arguments in function templates and in member functions of class
         templates are performed as described in 14.7.1."
Resolution:
Requestor:
Owner:          Steve Adamczyk (Default Arguments)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   689
Title:          What if two using-declarations refer to the same function but
                the declarations introduce different default-arguments?
Section:        8.3.6 [dcl.fct.default]
Status:         editorial
```

Description:

3.3 para 3 says:
"Given a set of declarations in a single declarative region, each of which specifies the same unqualified name,
-- they shall all refer to the same entity, or all refer to functions ..."

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

This is not really clear regarding what happens in the following case:

```
namespace A {
        extern "C" void f(int = 5);
}
namespace B {
        extern "C" void f(int = 7);
}

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

f(); // ???
```

Resolution:

At the Hawaii meeting, the core WG agreed that the example above was an error and suggested that this be clarified in the WP as an editorial matter.

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

. . . .

Work Group: Core
Issue Number: 866
Title: cv-qualifiers and type conversions
Section: 8.5 [dcl.init]
Status: active
Description:

1. The description of copy-initialization in 8.5 para 14 says:
   "The user-defined conversion so selected is called to convert the initializer expression into a temporary, whose type is the type returned by the call of the user-defined conversion function, with the cv-qualifiers of the destination type."

   Why must the temporary have the cv-qualifiers of the destination type? Shouldn't the cv-qualifiers of the conversion function dictate the cv-qualifiers of the temporary? For example,

```
struct A {
  A(A&);
};
struct B : A { };

struct C {
  operator B&();
};

C c;
const A a = c; // allowed?
```

   The temporary created with the conversion function is an lvalue of type B.

If the temporary must have the cv-qualifiers of the destination type (i.e. const) then the copy-constructor for A cannot be called to create the object of type A from the lvalue of type const B.

If the temporary has the cv-qualifiers of the result type of the conversion function, then the copy-constructor for A can be called to create the object of type A from the lvalue of type const B.

This last outcome seems more appropriate.

2. the treatment of cv-qualifiers in 13.3.1.4 is also puzzling:

   "Assuming that cv1 T is the type of the object being initialized
    ...
    --When the type of the initializer expression is a class type "cv S", the conversion functions of S and its base classes are considered. Those that are not hidden within S and yield type "cv2 T2", where T2 is the same type as T or is a derived class thereof, and where cv2 is the same cv-qualification as, or lesser cv-qualification than, cv1, are candidate functions."

   Why must the result of the conversion function be equally or less cv-qualified than the object initialized? Shouldn't the cv-qualification of the copy-constructor parameter determine whether the cv-qualification on the result of the conversion function is appropriate or not? For example:

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

struct C {
  operator const B&();
};

C c;
A a = c;
```

   The conversion function returns an lvalue of type const B. Shouldn't this be allowed since the copy constructor for class A accepts arguments that are const lvalues?

3. Is subclause 13.3.1.5 only for the initialization of non-class objects?

   The wording in this clause makes this somewhat confusing.
   The bullet in paragraph 1 says:
   "Conversion functions that return a nonclass type "cv2 T" are considered to yield cv-unqualified T for this process of selecting candidate functions."

   All the conversion functions considered in this section return "nonclass type". In which case, all the bits about cv-qualifiers are not necessary (and are somewhat confusing).

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

. . . .

==

Chapter 9 - Classes

```
--------------------
Work Group:     Core
Issue Number:   897
Title:          Can a union member be inspected through another member with
                the same "common initial sequence"?
Section:        9.5 [class.union]
Status:         editorial
Description:
        The ISO C standard in 6.3.2.3 "Structure and union members"
        describes the semantics of accessing union members.  The C++
        standard moves the descriptions around to at least three
        different places: 5.2.5 "Class member access", 9.2 "Class
        members", and 9.5 "Unions".

        The C standard says that if you retrive a value from a union
        from a member other than that used to store the value, the
        results are implementation-defined.  It goes on to make an
        exception for common initial sequences of structure members.

        The C++ draft has the "common initial sequence" language in
        9.2/16, but doesn't seem to have any other statement about
        accessing data via a member other than the one use to store
        data.

        The first sentence of 9.5 says "In a union, at most one of the
        data members can be active at any time, that is, the value of at
        most one of the data members can be stored in a union at any
        time." It could possibly be interpreted to cover the case in
        question, but in that case would mean the results are undefined,
        not implementation-defined.
Proposed Resolution:
        9.5 should probably say something similar to what the C standard says
        in 6.3.2.3:
        "One special guarantee is made in order to simplify the use of
         unions: If a union contains several structures that share a common
         initial sequence (_class.mem_), and if the union object currently
         contains one of these structures, it is permitted to inspect the
         common initial sequence if the corresponding members have
         layout-compatible types (and, for bitfields, the same widths) for a
         sequence of one or more initial members."
Resolution:
Requestor:      Steve Clamage
Owner:          Josee Lajoie (Object Model)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
==============================================================================
 Chapter 10 - Derived classes
------------------------------
==============================================================================
 Chapter 11 - Member Access Control
------------------------------------
Work Group:     Core
Issue Number:   872
Title:          How do access control apply to constructors/destructors
                implicitly called for static data members?
Section:        11 [class.access]
Status:         active
Description:
        Here's a question that is being discussed in comp.std.c++ for which I
        don't find a clear answer in the draft.

        class C {   // has private constructor and destructor
            friend class D;
```

```
        C();
        ~C();
    };

    class D {
    public:
        static C c; // static member
    };

    C D::c; // can this be constructed, and if so, can it be
            // destroyed?
```

Members of D can create and destroy objects of type C because the ctor and dtor are accessible. What about the static C member of D? Is its construction and destruction in the scope of D (accessible) or in global scope (inaccessible)? Where is the answer defined in the draft?

[Josee: Possible solution:]
Change 11 para 5 to the following:

All access controls in this clause affect the ability to access a class member name from a particular scope.
[addition:]
The access control for names used in the definition of a class member that appears outside of the member's class definition is done as if the entire member definition appeared in the scope of the member's class.
[end addition]
In particular, access controls apply as usual to member names accessed as part of a function return type, even though it is not possible to determine the access privileges of that use without first parsing the rest of the function.
[example in para 5]
[addition:]
Similarly, access control for the implicit calls to the constructor, conversion functions and destructor called to create and destroy such a class member is performed as if these calls appeared in the scope of the member's class.
[add the example above]

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

. . . .

Work Group: Core
Issue Number: 873
Title: How/when is access checked in default arguments of function templates?
Section: 11 [class.access]
Status: editorial
Description:

The following bit of text in the WP does not take into account the resolution adopted in London regarding default arguments of function templates.

11 para 7:
"The names in a default argument expression (8.3.6) are bound at the point of declaration, and access is checked at that point rather than at any points of use of the default argument expression."

Proposed Resolution:

I would like to add to the text above to say that these rules do not apply to default arguments in template functions and refer to 14.7.1 where the rules for default arguments are. How about the following,

```
        as a note:

        "Access checking for default arguments in function templates and in
         member functions of class templates are performed as described in
         14.7.1."
Resolution:
Requestor:
Owner:          Steve Adamczyk (Access)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   898
Title:          Access to template arguments used in a function return type
                and in the nested name specifier
Section:        11 [class.access]
Status:         active
Description:
        Consider the following example:

        class A {
            class A1{};
            static void func(A1, int);
            static void func(float, int);
            static const int garbconst = 3;
        public:
            template < class T, int i, void (*f)(T, int) > class int_temp {};
            template<> class int_temp<A1, 5, func> { void func1() };
            friend int_temp<A1, 5, func>::func1();
            int_temp<A1, 5, func>* func2();
        };

        A::int_temp<A::A1, A::garbconst + 2, &A::func>* A::func2() {...}

        ISSUE 1:
        ========
        In clause 11 we have:
        "5 All access controls in this clause affect the ability to
           access a class member name from a particular scope.  In
           particular, access con- trols apply as usual to member names
           accessed as part of a function return type, even though it is
           not possible to determine the access privileges of that use
           without first parsing the rest of the function."

        This means, if we take the loosest possible definition of
        "access from a particular scope", that we have to save and check
        later the following names
          A::int_temp
          A::A1
          A::garbconst (part of an expression)
          A::func (after overloading is done)

        I suspect that member templates were not really considered when
        this was written, and that it might have been written rather
        differently if they had been.  Note that access to the template
        arguments is only legal because the class has been declared a friend,
        which is probably not what most programmers would expect.

        ISSUE 2:
        ========
        Now consider

        void A::int_temp<A::A1, A::garbconst + 2, &A::func>::func1() {...}
```

By my reading of 11.8[class.access.nest], the references to A::A1, A::garbconst and A::func are now illegal, and there is no way to define this function outside of the class.

Is there any need to do anything about either of these Issues?

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

. . . .

Work Group: Core
Issue Number: 888
Title: Can a class with a private virtual base class be derived from?
Section: 11.2[class.access.base]
Status: active
Description:

```
class Foo { public: Foo() {}  ~Foo() {} };
class A : virtual private Foo { public: A() {}  ~A() {} };
class Bar : public A { public: Bar() {}  ~Bar() {} };
```

~Bar() calls ~Foo(), which is ill-formed due to access violation, right? (Bar's constructor has the same problem since it needs to call Foo's constructor.) There seems to be some disagreement among compilers. Sun, IBM and g++ reject the testcase, EDG and HP accept it. Perhaps this case should be clarified by a note in the draft.

In short, it looks like a class with a virtual private base can't be derived from.

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

. . . .

Work Group: Core
Issue Number: 899
Title: Clarification of access to base class members
Section: 11.2[class.access.base]
Status: active
Description:

11.2 para 4 says:
"A base class is said to be accessible if an invented public member of the base class is accessible. If a base class is accessible, one can implicitly convert a pointer to a derived class to a pointer to that base class."

Given the above, is the following well-formed?

```
class D;

class B
{
   protected:
       int b1;

   friend void foo( D* pd );
};

class D : protected B { };

void foo( D* pd )
{
```

```
    if ( pd->b1 > 0 ); // Is 'b1' accessible?
}
```

Can you access the protected member b1 of B in foo?
Can you convert a D* to a B* in foo?

1st interpretation:

A public member of B is accessible within foo (since foo is a friend), therefore foo can refer to b1 and convert a D* to a B*.

2nd interpretation:

B is a protected base class of D, and a public member of B is a protected member of D and can only be accessed within members of D and friends of D. Therefore foo cannot refer to b1 and cannot convert a D* to a B*.

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

. . . .

Work Group: Core
Issue Number: 900
Title: Can a nested class access its own class name as a qualified name if it is a private member of the enclosing class?
Section: 11.8 [class.access.nest]
Status: active
Description:

para 1 says: "The members of a nested class have no special access to members of an enclosing class..."

Does this prevent a member of a nested class from being defined outside of its class definition?
i.e. Should the following be well formed?

```
class D {
  class E {
    static E* m;
  };
};

D::E* D::E::m = 1; // well-formed?
```

In the draft standard, however, it isn't. This is because the nested class does not have access to the member E in D.

11 paragraph 5 says that access to D::E is checked with member access to class E, but unfortunately that doesn't give access to D::E.
11 paragraph 6 covers the access for D::E::m, but it doesn't affect the D::E access.

Are there any implementation that are standard compliant regarding this?

Here is another example:

```
class C {
  class B
  {
    C::B *t; //2 error, C::B is inaccessible
  };
};
```

This causes trouble for member functions declared outside of the class member list. For example:

```
class C {
  class B
  {
    B& operator= (const B&);
  };
};

C::B& C::B::operator= (const B&) { } //3
```

If the return type (i.e. C::B) is access checked in the scope of class B (as implied by 11 para 5) as a qualified name, then the return type is an error just like referring to C::B in the member list of class B above (i.e. //2) is ill-formed.

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

. . . .

==

Chapter 12 - Special Member functions

Work Group: Core
Issue Number: 901
Title: When is a temporary bound to a reference that is a local static variable destroyed?
Section: 12.2 [class.temporary]
Status: active
Description:

The wording in 12.2p5 says that a temporary to which a reference is bound "persists for the lifetime of the reference or until the end of the scope in which the temporary is created."

I think this does not properly address the case where the reference is a local static variable:

```
void f () {
        static const A &r = A();
}
```

I think the temporary in that case should persist as long as the reference does, period.

Also, since the case of binding a reference to a temporary in a ctor-initializer list and in a return statement is explicitly discussed, is the "or until the end of the scope in which the temporary is created" needed?

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

. . . .

Work Group: Core
Issue Number: 902
Title: When is 'template<class T> S(T);' used to generated a copy constructor?
Section: 12.8 [class.copy]
Status: editorial
Description:

```
        12.8 para 3 says:
        "A declaration of a constructor for a class X is ill-formed if its
         first parameter is of type (optionally cv-qualified) X and either
         there are no other parameters or else all other parameters have
         default arguments."

        What about the following example, does it use the template to
        generate the copy constructor?
        struct S {
          template<typename T> S(T);
        };

        S f();

        void g() {
          S a(f()); // OK?
        }

        John Spicer replied the following:
        > I think the intent of 12.8 paragraph 3 applies. This paragraph says
        > you can't declare a constructor like S(S).  We should probably make
        > an explicit statement that an S(S) function will not be generated
        > from a template to copy an object because it would just end up
        > calling itself to initialize its parameter.
Resolution:
Requestor:      David Vandervoorde
Owner:          Josee Lajoie (Object Model)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   876b
Title:          Should the optimization that allows a class object to alias
                another object also allow the case of a parameter in an
                inline function aliasing its argument?
Section:        12.8 [class.copy]
Status:         active
Description:
        At the London meeting, 12.8 [class.copy] paragraph 15 was changed
        to limit the optimization described to only the following cases:
        -- the source is a temporary object
        -- the return value optimization

        One other case was deemed desirable as well:
        -- aliasing a parameter in an inline function call to the function
           call argument.
        However, there are cases when this aliasing was deemed undesirable
        and, at the London meeting, the committee was not able to clearly
        delimit which cases should be allowed and which ones should be
        prohibited.

        Can we find an appropriate description for the desired cases?
Resolution:
Requestor:
Owner:          Josee Lajoie (Object Model)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
================================================================================
 Chapter 13 - Overloading
-------------------------
Work Group:     Core
Issue Number:   903
Title:          Is a function not viable if there exists two equally good
```

conversion sequences to convert an argument to the parameter type?
Section: 13.3.3.2 [over.ics.rank]
Status: active
Description:

```
class string
{
  private:
    class Dummy {};
    operator Dummy * () const;   //-- undefined
    //-- This dummy conversion function shall force compile
    //-- time errors when the class string is used in such
    //-- silly constructs as :
    //
    //--   string str("foo");
    //--   if (str) delete str;

  public:
    operator const char * () const { ... }
    ...
};

main()
{
    string str ("foo");
    cout << str;          //-- ambiguity ?
}
```

For the "cout << str" call, there is two viable functions :

```
ostream::operator<< (const char *)       func1
ostream::operator<< (const void *)       func2
```

For the first one, the implicit conversion sequence is :

```
string --> const char *       (user-defined)
```

For func2, two implicit conversion sequences exist :

```
string --> const char * -> const void *    (user-defined)
string --> Dummy * -> const void *         (user-defined)
```

Neither is better than the other (two user-defined sequences can be compared only if they use the same conversion function. See [over.ics.rank] para 3).

In this case, the compiler shall pick one of them randomly and, if the viable function that use it (func2) is found as the best, the call will be ill-formed (see [over.best.ics] para 10).

Scenario 1 : the compiler picks the first conversion sequence
----

To find the best viable function, the compiler tries to compare the implicit conversion sequences for all arguments (see [over.match.best]).

In this case, ICS1(func1) is not worse than ICS1(func2) and ICS2(func1) is better than ICS2(func2) :

```
ICS2(func1) = string --> const char *
ICS2(func2) = string --> const char * -> const void *
```

According to [over.ics.rank] para 3, one user-defined sequence

is better than another if :
-- they both use the same conversion function (that's the case)
-- the second standard conversion sequence of the former has a better rank than the latter (here, we have "exact match" vs. "conversion").

So, func1 is the better than func2. The call is not ill-formed because func2 has not been selected (see [over.best.ics] para 10).

Scenario 2 : the compiler picks the second conversion sequence
----

In this case, ICS2(func1) is neither better nor worse than ICS2(func2) because they don't use the same conversion function.

Therefore, func1 is neither better nor worse than func2
=> the call is ill-formed.

[Steve Adamczyk's reply:]

The right answer there is ambiguity.

That this is the intent is made clear by the footnote to 13.3.3.1 [over.best.ics] paragraph 10:

123) This rule prevents a function from becoming non-viable because of an ambiguous conversion sequence for one of its parameters. Consider this example,

```
class B;
class A { A (B&); };
class B { operator A (); };
class C { C (B&); };
void f(A) { }
void f(C) { }
B b;
f(b);   // ambiguous since b -> C via constructor and
        // b -> A via constructor or conversion function.
```

If it were not for this rule, f(A) would be eliminated as a viable function for the call f(b) causing overload resolution to select f(C) as the function to call even though it is not clearly the best choice. On the other hand, if an f(B) were to be declared then f(b) would resolve to that f(B) because the exact match with f(B) is better than any of the sequences required to match f(A).

The intent of the wording in paragraph 10 was to keep enough information about the ambiguous conversion to be able to compare it to the other possibilities, without keeping all the information on all the conversions. We had thought that picking one of the conversion sequences arbitrarily would work, but I don't think we considered cases like this one where some other conversion would have the same rank and where the specific conversion would matter.

There are two possible solutions that come to mind:

(a) For an ambiguous conversion sequence, keep all of the possible conversion sequences, so they can all be compared individually against the conversion sequences for other candidates. This is potentially onerous, but it certainly wouldn't be the first such requirement in C++. :-)

```
        (b) For an ambiguous conversion sequence, keep only the rank and
        the fact that it's ambiguous.  When such an ambiguous conversion
        sequence is compared to another conversion sequence, it could be
        judged better or worse on the basis of rank, but it would be no
        better and no worse than any conversion sequence with the same
        rank.  (This latter is effectively what EDG implements.)

        [Bill Gibbons]:
        I think we also need an editorial change to 13.3.2/3:

           Second, for F to be a viable function, there shall exist for
           each argument an implicit conversion sequence (13.3.3.1) that
           converts that argument to the corresponding parameter of F.

        such as another sentence:

           The conversion sequence need not be unique; see _over.best.ics_.
Resolution:
Requestor:      Jerome Charousset (via Andrew Koenig)
Owner:          Steve Adamczyk (Overload Resolution)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   889
Title:          pseudo prototypes for built-in operators and operands of
                enumeration types need fine tuning
Section:        13.6 [over.built]
Status:         active
Description:
        Issue 1:

        Here's a program that was formerly valid, and now gets an
        ambiguity error:

          enum E {E1};

          struct A {
              A();
              A(E);
              friend int operator==(A, E);
          };

          int main()
          {
              E e;
              A a;
              e == E1;  // Now ambiguous
          }

        The problem is that the 13.6 pseudo-prototypes for the "=="
        operator (and many others) do not explicitly deal with enums.
        As a consequence, any time an enum expression participates in an
        operation, it has to undergo at least a promotion to get to an
        arithmetic type.  In the above example, that means the built-in
        operator "==" is worse than the friend function on the second
        operand.  Since the built-in operator is better on the first
        operand, the case is ambiguous.

        Issue 2:

        This is a case that wasn't valid previously (because it
        declares an operator function with an enum parameter and no
```

class parameter), but which gets a surprising answer:

```
enum E {E1};

E operator+(E,int);

int main()
{
    E e;
    e + E1;  // Uses ::operator+
}
```

Case 2 seems less serious to me than case 1, partly because addition is not an operation on enums. I think adding two enums can reasonably be interpreted as going through the integral promotions before the addition is done.

Proposed Resolution:
The solution is probably to add more pseudo-prototypes in 13.6 to deal with the case where the operands of a builtin operation have the same enum type. This is particularly important for comparison operators, and for the "?" operator (but there is already an open core issue for that one).
Resolution:
Requestor: Steve Adamczyk
Owner: Steve Adamczyk (Overload Resolution)
Emails:
Papers:

. . . .

Work Group: Core
Issue Number: 904
Title: The prototypes for ?: must be fixed now that lvalue-to-rvalue was removed
Section: 13.6 [over.built]
Status: active
Description:
I understand that the lvalue-to-rvalue conversion was removed in London. I generally agree with this, but it means that ?: needs to be fixed. Given

```
bool test;
Integer a, b;
test ? a : b;
```

What builtin do we use? The candidates are

```
operator ?:(bool, const Integer &, const Integer &) <builtin>
operator ?:(bool, Integer, Integer) <builtin>
```

which are both perfect matches.
Resolution:
Requestor: Jason Merrill
Owner: Steve Adamczyk (Overload Resolution)
Emails:
Papers:

. . . .

==

Chapter 14 - Templates

Work Group: Core
Issue Number: 781
Title: Must default template-arguments be provided only on the first template declaration?
Section: 14.1 [temp.param]

Status: active
Description:

14.1 paragraph 8 says the following:
"The set of default template-arguments available for use with a template in a translation unit shall be provided only by the first declaration of the template in that translation unit."

This is causing some trouble to the library WG.

John Spicer noted the following:
> There is a good reason for this rule (or a rule like this) for
> function templates. It gets messy if you permit default arguments
> to be added after the template has been referenced.
>
> There is not a good reason for the rule for classes. The WP
> inadvertantly got changed to have this rule apply to classes, and
> we decided not to change it back because we thought the restriction
> was harmless. The previous rule for classes was the same as the
> usual rules for nontemplate functions (i.e., that you can't
> redeclare a default argument but you can add one). Presumably,
> this would fix the library problem as the default argument could be
> placed on the definition of the class, and not on any of the
> forward declarations.

Should this be revisited?

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

. . . .

Work Group: Core
Issue Number: 765
Title: The syntax does not allow the keyword 'template' where the text in 14.2 says it is allowed
Section: 14.2 [temp.names]
Status: active
Description:

The current C++ grammar does not support the use of the template keyword in all the places where subclause 14.2 says it is allowed. For example, the following cases are not allowed by the grammar:

In qualified-ids:

```
    A<T>::template B<X>::template C<Y>
          -------------
```

In pseudo-destructor-calls:

```
    p->A::template B<T>::~B();
          -------------
```

After discussions with Bill Gibbons, John Spicer, Anthony Scian and myself, it seems that we cannot come to an agreement as to how to fix this.

Here are the two approaches that are under consideration:

1) allow the template keyword in the template-name production, i.e.

```
   template-name
       template(opt) identifier
```

This is a simple grammar fix but it allows the 'template' keyword in many more contexts than that currently allowed by chapter 14. The solution would be to prohibit the 'template' keyword to appear in these additional contexts by adding additional semantics rules in the WP.

2) apply the grammar change higher up in the grammar, to allow the keyword template only in the places that are already allowed by the text in chapter 14. This means that a greater number of grammar rules must be changed and there is the possibility that we did not cover all cases.

Paper 97-0085/N1123 in the pre-Morristown mailing outlines the possible solutions in greater detail.

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

. . . .

Work Group: Core
Issue Number: 905
Title: How does a template template argument that is a partial specialization match a template template parameter?
Section: 14.3.3 [temp.arg.template]
Status: active
Description:

If a template template argument is a partially specialized class template, what are the rules for matching it with a template template parameter? Can a partial specialization match? If so, are the ordering rules used to disambiguate when more than one variant matches?

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

. . . .

Work Group: Core
Issue Number: 906
Title: Does the 'this' pointer of conversion function member templates participate in overload resolution?
Section: 14.5.2[temp.mem]
Status: editorial
Description:

Para 5 says:
"If more than one conversion template can produce the required type, the partial ordering rules (_temp.func.order_) are used to select the "most specialized" version of the template that can produce the required type. As with other conversion functions, the type of the implicit this parameter is not considered."

However, 13.3.1.5 [over.match.conv] para 2 seems to contradict this:
"The argument list has one argument, which is the initializer expression. [Note: this argument will be compared against the implicit object parameter of the conversion functions.]

Steve Adamczyk replied the following:

> The sentence:
> "As with other conversion functions, the type of the implicit this
> parameter is not considered."
> is not intended to be a statement about overload resolution, but
> rather about partial ordering, and should probably read something
> like:
> "As with other conversion functions, the type of the implicit this
> parameter does not affect the determination of partial ordering".
> Furthermore, this is really a restatement of how one aspect of
> partial ordering works, and should be moved into the following
> note.

```
Resolution:
Requestor:      Jason Merrill
Owner:          Bill Gibbons (Templates)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   890
Title:          Clarification of the interaction of friend declarations
                and use of explicit template arguments
Section:        14.5.3[temp.friend]
Status:         active
Description:
        Issue 1:
        ========
        Can a friend declaration for which the declarator is a qualified-id
        refer to a template specialization even though explicit template
        arguments are not specified?

        For example, does the friend declaration in A make an instance
        of N::f a friend?

        namespace N {
                template <class T> void f(T);
        }

        template <class T> struct A {
                friend void N::f(T);
        };

        John Spicer's answer:
        > It should be a valid means of making an instance of N::f a
        > friend.  Only unqualified friend declarations should be
        > prohibited from referring to a previously declared template
        > unless explicit template arguments are used.  Our rationale
        > for this is:
        >
        > 1. It is consistent with the way in which functions are
        >    called.  An explicit template argument list is only needed
        >    in a call when the user wants to force the compiler to use
        >    a template.  In the absence of an explicit template
        >    argument list, overload resolution (for a call) or type
        >    matching (for the address of a function) is used to select
        >    the best match.
        >
        > 2. The real need is to guarantee that an unqualified
        >    declaration introduces a new function, and does not refer
        >    to the template.  Permitting qualified references to
        >    previously declared templates in no way compromises this.
        >
        > 3. It eliminates a gratuitious incompatibility with existing
        >    code.

        Issue 2:
        ========
        -- How is f looked up in the following friend declaration?

        template <class T> void f(T) {}
        struct A {
                friend void f<int>(int);
        };

        John Spicer's proposal:
        > It is looked up using the normal lookup rules for unqualified name
```

```
        > specified in 3.4.1. The example above is a reference to the name
        > "f".

        -- How are f and g looked up in the friend declarations?

        namespace N {
                template <class T> void f(T);
                void g(int);
                namespace O {
                        struct A {
                                friend void f<int>(int); // N::f
                                friend void g(int); // declares O::g
                        };
                }
        }

        John Spicer's proposal:
        > A name declared in a friend declaration is a member of the nearest
        > enclosing namespace and the search for a previous declaration
        > extends only as far as that namespace.  In the example above, there
        > is no reason for the "friend void f<int>(int)" declaration not to
        > find the template declared in namespace N. The "friend void g(int)"
        > declaration, on the other hand, declares O::g because the search
        > for a previous declaration does not extend to namespace N.
        >
        > The one unfortunate consequence of this rule is that special care
        > needs to be taken when a friend declaration with an explicit
        > template argument list refers to a name that is also a member of
        > the current class. In such cases, a qualified name must be used in
        > order to refer to the template from the outer scope.
        >
        > template <class T> void f(T) {}
        > struct A {
        >         template <class T> void f(T) {}
        >         friend void ::f<int>(int);
        > };
Requestor:      John Spicer
Owner:          Bill Gibbons (Templates)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   907
Title:          How can a partial specialization be used by the definition of
                an exported template?
Section:        14.5.4 [temp.class.spec]
Status:         active
Description:
        The resolution to ballot comment issue "L7052 USA Core3 1.19"
        is impractical; this issue must be revisited.
        The problem is that it is not generally possible to provide
        a partial specialization of a template in the context of the
        definition. See editorial box in 14.5.4.
Resolution:
Requestor:
Owner:          Bill Gibbons (Templates)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   908
Title:          Syntax for partial specialization missing
Section:        14.5.4 [temp.class.spec]
Status:         active
```

```
Description:
        I happened to trace through the grammar looking for the syntax
        for partial specializations; to my surprise, I discovered that
        it appears not to be there!  That is, the only things that can
        follow a class-key are qualified and unqualified identifiers,
        not template-ids.  Thus, according to the current grammar,
        something like

                template <class T, class U>     class C { };
                template <class T>              class C<T, int> { };

        is a syntax error.

        This seems to me to be sufficiently broken that it should be
        fixed before DIS.
Resolution:
Requestor:      Mike Miller
Owner:          Bill Gibbons (Templates)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   882
Title:          typename is not permitted in functional cast notation
Section:        14.6 [temp.res]
Status:         active
Description:
        The use of typename in a function-style cast was agreed on but
        did not make it into the motions (core 882):

          template <class T> int f(T) {
              return typename T::inner(); // typename should be allowed
          }

        where "T::inner" is required to be a type.

        [Bill Gibbons:]
        > In London, core-III voted to recommend that the "typename" keyword
        > be allowed in function-style casts. I now think that this change
        > should NOT be made.
        >
        > Given that neither elaborated-type-specifiers nor multi-keyword
        > type names are allowed in function-style casts, it does not seem
        > appropriate to allow "typename" either.
        >
        > template <class T> int f(T) {
        >       return class T::inner(); // ill-formed
        > }
        > long int f(int x) { return long int(x); } // ill-formed
        >
        > Such casts can always be written using new-style or C-style casts.

        [Matt Austern]:
        > The real issue isn't function-style cases, I think, but constructor
        > calls.
        >
        > As Bill points out, there's a workaround for the fact that
        > function-style casts don't work: using static_cast instead.  I
        > don't think there's a workaround for constructors, though.
        >
        > template <class T>
        > typename T::Pair_Type foo(T) {
        >   return typename T::Pair_Type(1, 2);
        > }
```

```
        >
        > If I understand the issue correctly, foo() is ill formed according
        > to the latest WP.  That seems to be a bad thing.
        >
        > I don't think that it's valid to equate "typename T::Pair_Type"
        > with "class X".  In the latter case, the "class" keyword is an
        > unnecessary elaboration.  In the former, though, there is no way
        > to refer to the type T::Pair_Type without the "typename" keyword.
        > It's unreasonable for the language to simulaneously require and
        > prohibit "typename".
Resolution:
Requestor:      John Spicer
Owner:          Bill Gibbons (Templates)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   909
Title:          Is the unqualified name of a partial specialization
                implicitly followed by template arguments in its own class
                scope?
Section:        14.6.1 [temp.dep.res]
Status:         editorial
Description:
        Para 1 says that within a class template, the name of that template
        is really equivalent to that name followed by the template
        parameter-list in angle brackets.

        However, partial specializations are not covered adequately:

        template<typename T>
        struct Node<T*> {
           T *data_;
           Node *next_; // really: Node<T> *next_; ??
        };              // not: Node<T*> *next_;

        [John Spicer:]
        > I think we can have one rule that covers both primary templates and
        > partial specializations.  14.5.4 [temp.class.spec] describes the
        > "template argument list" associated with a class template primary
        > declaration or partial specialization.  A primary template has an
        > implicit template argument list that is simply its template
        > parameters named in order.  A partial specialization's template
        > argument list is the one that is specified after the name of the
        > template. i.e.
        >
        > template <class T, int I> struct A {};  // implicit <T,I>
        > template <class T> struct A<T*, 5> {}; // explicit <T*,5>
        >
        > So, instead of saying that the name of the template is equivalent
        > to name<temlate-parameters>, we would say that it is equivalent to
        > name<template-argument-list>, where template-argument-list is the
        > template argument list described in 14.5.4.
Resolution:
Requestor:      David Vandervoorde
Owner:          Bill Gibbons (Templates)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   737
Title:          How can dependant names be used in member declarations
                that appear outside of the class template definition?
Section:        14.6.4 [temp.dep.res]
```

```
Status:         editorial
Description:
         template <class T> class Foo {
           public:
           typedef int Bar;
           Bar f();
         };
         template <class T> typename Foo<T>::Bar Foo<T>::f() { return 1;}
                            --------------------

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

           int Foo<T>::f();

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

         From John Spicer:
         > My opinion (which I think matches several posted on the
         > reflector recently) is that the out-of-class definition must
         > match the declaration in the template.  In your example they
         > do match, so it is well formed.
         >
         > I've added some additional cases that illustrate cases that
         > I think either are allowed or should be allowed, and some
         > cases that I don't think are allowed.
         >
         > template <class T> class A { typedef int X; };
         >
         > template <class T> class Foo {
         > public:
         >   typedef int Bar;
         >   typedef typename A<T>::X X;
         >   Bar f();
         >   int g1();
         >   Bar g2();
         >   X h();
         >   X i();
         >   int j();
         > };
         >
         > // Declarations that are okay
         > template <class T> typename Foo<T>::Bar Foo<T>::f()
         >                                                  { return 1;}
         > template <class T> typename Foo<T>::Bar Foo<T>::g1()
         >                                                  { return 1;}
         > template <class T> int Foo<T>::g2() { return 1;}
         > template <class T> typename Foo<T>::X Foo<T>::h() { return 1;}
         >
         > // Declarations that are not okay
         > template <class T> int Foo<T>::i() { return 1;}
         > template <class T> typename Foo<T>::X Foo<T>::j() { return 1;}
         >
         > In general, if you can match the declarations up using only
         > information from the template, then the declaration is valid.
         >
```

```
         > Declarations like Foo::i and Foo::j are invalid because for
         > a given instance of A<T>, A<T>::X may not actually be int if
         > the class is specialized.
         >
         > This is not a problem for Foo::g1 and Foo::g2 because for
         > any instance of Foo<T> that is generated from the template
         > you know that Bar will always be int. If an instance of Foo
         > is specialized, the template member definitions are not used
         > so it doesn't matter whether a specialization defines Bar as
         > int or not.
```

Resolution:

When a member function of a class template is defined outside the class, and the return type is specified by a member of a dependent class, the typename keyword is needed to specify that the member name is a type. So the typename keyword should be allowed in this context.

Core 3 agreed that this is largely editorial.
Some work is needed to figure out exactly what needs to be said.

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

. . . .

Work Group: Core
Issue Number: 910
Title: Which part of the class member list is instantiated when a class template is instantiated?
Section: 14.7.1 [temp.inst]
Status: active
Description:

14.7.1 does not describe clearly which part of a class member defined within its class definition is instantiated when a class template is instantiated. For example, is the following ill-formed when a1 is defined because there are two member functions address(const T &) declared in the class member list, or is this only an error if the member function address is called?

```
template<class T>
struct allocator {
  void address(T& ) { }
  void address(const T& ) { }
};

allocator< const int > a1;
```

Another example:

```
struct A{
  //typedef int I;
};
template<class T> class X {
  typename T::I f() {}
};

X<A> a1;
```

Is the above ill-formed when a1 is defined because there are no type named I in class A or is the above ill-formed only when f is called?

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

. . . .

```
Work Group:     Core
Issue Number:   839
Title:          The template compilation model rules render some explicit
                specialization declarations not visible during instantiation
Section:        14.7.3 [temp.expl.spec]
Status:         active
Description:
        [N1065 issue 1.19]
        An explicit specialization declaration may not be visible during
        instantiation under the template compilation model rules, even though
        its existence must be known to perform the instantiation correctly.
        For example:

        translation unit #1
          template<class T> struct A { };
          export template<class T> void f(T) { A<T> a; }

        translation unit #2
          template<class T> struct A { };
          template<> struct A<int> { }; // not visible during instantiation
          template<class T> void f(T);
          void g() { f(1); }
Resolution:
Requestor:      Bill Gibbons
Owner:          Bill Gibbons (Templates)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   911
Title:          What happens if the explicit template arguments for an
                overloaded function template only match some of the
                variants?
Section:        14.8.1 [temp.arg.explicit]
Status:         editorial
Description:
        There is no mention of what happens if explicit template
        arguments for an overloaded function template only match some of
        the variants:

          template<class T> void f();
          template<void *p> void f();
          void g() { f<int>(); }

        For the *implicit* template argument case, if deduction fails
        the template is simply not considered.  For the *explicit* case,
        the working paper implies that the program is ill-formed if any
        of the matching function templates cannot accept the explicit
        arguments.
Proposed Resolution:
        The non-matching function templates should just be ignored.
Resolution:
Requestor:
Owner:          Bill Gibbons (Templates)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   912
Title:          Template argument deduction and pointer to member function
                types
Section:        14.8.2.4 [temp.deduct.type]
Status:         editorial
Description:
```

```
        para 9 says:
        "where (T) represents argument lists where at least one argument type
         contains a T, and () represents argument lists where no parameter
         contains a T."

        The part of 'no parameter contains a T' does not hold for pointer to
        member functions.  The interpretation has mostly been 'no parameter
        needs to contain a T, but some may'.  This should either be said or
        reflected in the patterns in para 9.

        Bill Gibbons' proposed resolution:
        > I think the correct fix is to combine all the lines where there
        > are distinct types named, some of which are marked as being
        > dependent and some not, as in:
        >
        >    type (*) (T)
        >    T    (*) (T)
        >    T    (*) ( )
        >
        > into single lines of the form:
        >
        >    T (*) (T)
        >
        > and add text to the effect that:
        >
        > The notation (T) represents a (possibly empty) argument list which
        > may or may not depend on T.  For each of these forms, at least one
        > of the types (or parameter types) represented by T must contain
        > a T.
Resolution:
Requestor:      Erwin Unruh
Owner:          Bill Gibbons (Templates)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
==============================================================================
 Chapter 15 - Exception Handling
---------------------------------
Work Group:     Core
Issue Number:   913
Title:          What happens if a terminate() handler causes terminate() to
                be reinvoked?
Section:        15.5.1[except.terminate]
Status:         active
Description:
        Does the draft say anywhere what happens if a `terminate()' handler
        itself causes terminate() to be reinvoked?

        > No.  Nor does it say whether any exception handling at all can
        > occur while terminate() is executing.
        >
        > In the absence of any restrictions, then, terminate() can be
        > called recursively and the behavior seems to be well-defined.
        > [...]
        > Allowing recursive calls to terminate() may be undesirable.
Resolution:
Requestor:      David Vandervoorde
Owner:          Bill Gibbons (Exceptions)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Work Group:     Core
Issue Number:   847
Title:          The description of "unexpected" in 18.6.2.2 differs from
```

```
                15.5.2
Section:        15.5.2[except.unexpected]
Status:         editorial
Description:
Resolution:
        The description of "unexpected" in 18.6.2.2 para 2 differs from the
        description in 15.5.2. The description in 15.5.2 is correct; the one
        in 18.6.2.2 should either be changed to match or be replaced with a
        cross-reference to 15.5.2.
Requestor:
Owner:          Bill Gibbons (Exceptions)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
==============================================================================
 Chapter 16 - Preprocessing Directives
---------------------------------------
==============================================================================
 Annex C - Compatibility
-------------------------
==============================================================================
 Annex E - Universal-character-names
------------------------------------
Work Group:     Core
Issue Number:   770
Title:          The title of Annex E needs to be made shorter
Section:        Annex E[extendid]
Status:         editorial
Description:
        The top of page E-2 (Annex E) has the section title overlapping
        the date.

        Andrew Koenig responded the following:
        > The reason is that (major) clause titles aren't checked for
        > overlap with the date.  The easiest fix is therefore to
        > rename clause E to something shorter.
Resolution:
        The title of the annex should be changed.
        Possible candidate: "Universal-character-names".
Requestor:
Owner:          Tom Plum (Annex E)
Emails:
Papers:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
```