Date: 2018-04-11
Project: Programming Language C++
Reference: ISO/IEC IS 14882:2014
Reply to: William M. Miller
Edison Design Group, Inc.
wmm@edg.com

C++ Standard Core Language Defect Reports and Accepted Issues, Revision 100

This document contains the C++ core language issues that have been categorized as Defect Reports by the Committee (PL22.16 + WG21) and other accepted issues, that is, issues with status "DR," "accepted," "DRWP," "WP," "CD1," "CD2," "CD3," "CD4," "TC1," "C++11," "C++14," and "C++17," along with their proposed resolutions. **Issues with DR, accepted, DRWP, and WP status are NOT part of the International Standard for C++.** They are provided for informational purposes only, as an indication of the intent of the Committee. They should not be considered definitive until or unless they appear in an approved Technical Corrigendum or revised International Standard for C++.

This document is part of a group of related documents that together describe the issues that have been raised regarding the C++ Standard. The other documents in the group are:

- Active Issues List, which contains explanatory material for the entire document group and a list of the issues that have not yet been acted upon by the Committee.
- Closed Issues List, which contains the issues which the Committee has decided are not defects in the International Standard, including a brief rationale explaining the reason for the decision.
- Table of Contents, which contains a summary listing of all issues in numerical order.
- Index by Section, which contains a summary listing of all issues arranged in the order of the sections of the Standard with which they deal most directly.
- Index by Status, which contains a summary listing of all issues grouped by status.

For more information, including a description of the meaning of the issue status codes and instructions on reporting new issues, please see the Active Issues List.

Section references in this document reflect the section numbering of document WG21 N4700.

Issues with "DR" Status

2164. Name hiding and *using-directive*s

Section: 6.3.10 [basic.scope.hiding] **Status:** DR **Submitter:** Richard Smith **Date:** 2015-07-26

[Accepted at the March, 2018 (Jacksonville) meeting.]

Consider the following example:

```
const int i = -1;
namespace T {
  namespace N { const int i = 1; }
  namespace M {
    using namespace N;
    int a[i];
  }
}
```

According to 6.4.1 [basic.lookup.unqual], lookup for `i` finds `T::N::i` and stops. However, according to 6.3.10 [basic.scope.hiding] paragraph 1, the appearance of `T::N::i` in namespace `T` does *not* hide `::i`, so both declarations of `i` are visible in the declaration of `a`.

It seems strange that we specify this name hiding rule in two different ways in two different places, but they should at least be consistent.

On a related note, the wording in 6.4.1 [basic.lookup.unqual] paragraph 2, "as if they were declared in the nearest enclosing namespace..." could be confusing with regard to the "declared in the same scope" provisions of 6.3.10 [basic.scope.hiding].

Proposed resolution (November, 2017)

Change 6.3.10 [basic.scope.hiding] paragraphs 1 and 2 as follows:

> ~~A name can be hidden by an explicit declaration of that same name in a nested declarative region or derived class (13.2 [class.member.lookup])~~ **A declaration of a name in a nested declarative region hides a declaration of the same name in an enclosing declarative region; see 6.3.1 [basic.scope.declarative] and 6.4.1 [basic.lookup.unqual]**.
>
> ~~A class name (12.1 [class.name]) or enumeration name (10.2 [dcl.enum]) can be hidden by the name of a variable, data member, function, or enumerator declared in the same scope.~~ If a class **name (12.1 [class.name])** or enumeration name **(10.2 [dcl.enum])** and a variable, data member, function, or enumerator are declared in the same ~~scope~~ **declarative region** (in any order) with the same name **(excluding declarations made visible via *using-directive*s (6.4.1 [basic.lookup.unqual]))**, the class or enumeration name is hidden wherever the variable, data member, function, or enumerator name is visible.

1910. "Shall" requirement applied to runtime behavior

Section: 6.7.4.1 [basic.stc.dynamic.allocation] **Status:** DR **Submitter:** Richard Smith **Date:** 2014-04-12

[Accepted at the March, 2018 (Jacksonville) meeting.]

According to 6.7.4.1 [basic.stc.dynamic.allocation] paragraph 3,

> If an allocation function declared with a non-throwing *exception-specification* (18.4 [except.spec]) fails to allocate storage, it shall return a null pointer. Any other allocation function that fails to allocate storage shall indicate failure only by throwing an exception (18.1 [except.throw]) of a type that would match a handler (18.3 [except.handle]) of type `std::bad_alloc` (21.6.3.1 [bad.alloc]).

The use of the word "shall" to constrain runtime behavior is inappropriate, as it normally identifies cases requiring a compile-time diagnostic.

Proposed resolution (November, 2017)

1. Change 6.7.4 [basic.stc.dynamic] paragraph 3 as follows:

> ~~Any allocation and/or deallocation functions defined in a C++ program, including the default versions in the library, shall conform to the semantics~~ **If the behavior of an allocation or deallocation function does not satisfy the semantic constraints** specified in 6.7.4.1 [basic.stc.dynamic.allocation] and 6.7.4.2 [basic.stc.dynamic.deallocation]**, the behavior is undefined**.

2. Change 6.7.4.1 [basic.stc.dynamic.allocation] paragraph 1 as follows:

> ...The value of the first parameter ~~shall be~~ **is** interpreted as the requested size of the allocation...

3. Change 6.7.4.1 [basic.stc.dynamic.allocation] paragraph 2 as follows:

> ~~The~~ **An** allocation function attempts to allocate the requested amount of storage. If it is successful, it ~~shall return~~ **returns** the address of the start of a block of storage whose length in bytes ~~shall be~~ **is** at least as large as the requested size. ~~There are no constraints on the contents of the allocated storage on return from the allocation function.~~ The order, contiguity, and initial value of storage allocated by successive calls to an allocation function are unspecified. ~~The~~ **For an allocation function other than a reserved placement allocation function (21.6.2.3 [new.delete.placement], the** pointer returned ~~shall be~~ **is** suitably aligned so that it can be converted to a pointer to any suitable complete object type (21.6.2.1 [new.delete.single]) and then used to access the object or array in the storage allocated (until the storage is explicitly deallocated by a call to a corresponding deallocation function). Even if the size of the space requested is zero, the request can fail. If the request succeeds, the value returned ~~shall be~~ **by a replaceable allocation function is** a non-null pointer value (7.11 [conv.ptr]) `p0` different from any previously returned value `p1`, unless that value `p1` was subsequently passed to ~~an operator delete~~ **a replaceable deallocation function**. Furthermore, for the library allocation functions in 21.6.2.1 [new.delete.single] and 21.6.2.2 [new.delete.array], `p0` ~~shall represent~~ **represents** the address of a block of storage disjoint from the storage for any other object accessible to the caller. The effect of indirecting through a pointer returned ~~as~~ **from** a request for zero size is undefined.[38]

4. Change 6.7.4.1 [basic.stc.dynamic.allocation] paragraph 3 as follows:

> An allocation function that fails to allocate storage can invoke the currently installed new-handler function (21.6.3.3 [new.handler]), if any. [*Note:* A program-supplied allocation function can obtain the address of the currently installed `new_handler` using the `std::get_new_handler` function (21.6.3.4 [set.new.handler]). —*end note*] ~~If an~~ **An** allocation function that has a non-throwing exception specification (18.4 [except.spec]) ~~fails to allocate storage, it shall return~~ **indicates failure by returning** a null pointer **value**. Any other allocation function ~~that fails to allocate storage shall indicate~~ **never returns a null pointer value and indicates** failure only by throwing an exception (18.1 [except.throw]) of a type that would match a handler (18.3 [except.handle]) of type `std::bad_alloc` (21.6.3.1 [bad.alloc]).

2241. Overload resolution is not invoked with a single function

Section: 8.2.2 [expr.call] **Status:** DR **Submitter:** CWG **Date:** 2016-03-04

Various aspects of overload resolution, such as determining viable functions, specification of conversion sequences, etc., should apply when there is only one function in the overload set, but currently overload resolution is only invoked to choose among multiple functions.

Proposed resolution (March, 2018):

This issue is resolved by the resolution of issue 2092.

1893. Function-style cast with *braced-init-list*s and empty pack expansions

Section: 8.2.3 [expr.type.conv] **Status:** DR **Submitter:** Richard Smith **Date:** 2014-03-13

[Accepted at the March, 2018 (Jacksonville) meeting.]

According to 8.2.3 [expr.type.conv] paragraph 1,

> A *simple-type-specifier* (10.1.7.2 [dcl.type.simple]) or *typename-specifier* (17.7 [temp.res]) followed by a parenthesized *expression-list* constructs a value of the specified type given the expression list. If the expression list is a single expression, the type conversion expression is equivalent (in definedness, and if defined in meaning) to the corresponding cast expression (8.4 [expr.cast]). If the type specified is a class type, the class type shall be complete. If the expression list specifies more than a single value, the type shall be a class with a suitably declared constructor (11.6 [dcl.init], 15.1 [class.ctor]), and the expression `T(x1, x2, ...)` is equivalent in effect to the declaration `T t(x1, x2, ...);` for some invented temporary variable `t`, with the result being the value of `t` as a prvalue.

This does not cover the cases when the *expression-list* contains a single *braced-init-list* (which is neither an expression nor more than a single value) or if it contains no expressions as the result of an empty pack expansion.

Proposed resolution (June, 2014): [SUPERSEDED]

This issue is resolved by the resolution of issue 1299.

Proposed resolution (November, 2017)

Change 8.2.3 [expr.type.conv] paragraph 2 as follows:

> If the initializer is a parenthesized single expression, the type conversion expression is equivalent to the corresponding cast expression (8.4 [expr.cast]). Otherwise, if the type is *cv* `void` and the initializer is `()` **(after pack expansion, if any)**, the expression is a prvalue of the specified type that performs no initialization. Otherwise, the expression is a prvalue of the specified type whose result object is direct-initialized (11.6 [dcl.init]) with the initializer. ~~For an expression of the form `T()`, `T`~~ **If the initializer is a parenthesized optional *expression-list*, the specified type** shall not be an array type.

2338. Undefined behavior converting to short enums with fixed underlying types

Section: 8.2.9 [expr.static.cast] **Status:** DR **Submitter:** Thomas Köppe **Date:** 2017-03-24

[Accepted as a DR at the November, 2017 meeting.]

The specifications of `std::byte` (21.2.5 [support.types.byteops]) and `bitmask` (20.4.2.1.4 [bitmask.types]) have revealed a problem with the integral conversion rules, according to which both those specifications have, in the general case. undefined behavior. The problem is that a conversion to an enumeration type has undefined behavior unless the value to be converted is in the range of the enumeration.

For enumerations with an unsigned fixed underlying type, this requirement is overly restrictive, since converting a large value to an unsigned integer type is well-defined.

Proposed resolution (August, 2017):

Change 8.2.9 [expr.static.cast] paragraph 10 as follows:

> A value of integral or enumeration type can be explicitly converted to a complete enumeration type. ~~The~~ **If the enumeration type has a fixed underlying type, the value is first converted to that type by integral conversion, if necessary, and then to the enumeration type. If the enumeration type does not have a fixed underlying type, the** value is unchanged if the original value is within the range of the enumeration values (10.2 [dcl.enum]). ~~Otherwise~~**, and otherwise**, the behavior is undefined.

2342. Reference `reinterpret_cast` and pointer-interconvertibility

Section: 8.2.10 [expr.reinterpret.cast] **Status:** DR **Submitter:** Richard Smith **Date:** 2017-04-07

[Accepted as a DR at the November, 2017 meeting.]

The changes from document P0137 make it clear that this is valid:

```
struct A { int n; } a;
int *p = reinterpret_cast<int*>(&a); // ok, a and a.n are pointer-interconvertible
int m = *p;                          // ok, p points to a.n
```

but the handling for that case does not extend to this one:

```
int &r = reinterpret_cast<int&>(a);
int n = r;
```

The relevant rule is 8.2.10 [expr.reinterpret.cast] paragraph 11:

A glvalue expression of type `T1` can be cast to the type "reference to `T2`" if an expression of type "pointer to `T1`" can be explicitly converted to the type "pointer to `T2`" using a `reinterpret_cast`. The result refers to the same object as the source glvalue, but with the specified type. [*Note:* That is, for lvalues, a reference cast `reinterpret_cast<T&>(x)` has the same effect as the conversion `*reinterpret_cast<T*>(&x)` with the built-in `&` and `*` operators (and similarly for `reinterpret_cast<T&&>(x)`). —*end note*]

Note that the normative rule and the note specify different rules: under the rule described in the note, the result would be a reference to the object `a.n`. According to the normative rule, however, we get a reference to the object `a` with type `n`.

Proposed resolution (July, 2017):

Change 8.2.10 [expr.reinterpret.cast] paragraph 11 as follows:

A glvalue expression of type `T1`**, designating an object** `x`**,** can be cast to the type "reference to `T2`" if an expression of type "pointer to `T1`" can be explicitly converted to the type "pointer to `T2`" using a `reinterpret_cast`. The result ~~refers to the same object as the source glvalue, but with the specified type~~ **is that of** `*reinterpret_cast<T2 *>(p)` **where** `p` **is a pointer to** ***x*** **of type "pointer to** `T1`**"**. ~~[*Note:* That is, for lvalues, a reference cast `reinterpret_cast<T&>(x)` has the same effect as the conversion `*reinterpret_cast<T*>(&x)` with the built-in `&` and `*` operators (and similarly for `reinterpret_cast<T&&>(x)`). —*end note*]~~ No temporary is created, no copy is made, and constructors (15.1 [class.ctor]) or conversion functions (15.3 [class.conv]) are not called. [*Footnote:* This is sometimes referred to as a *type pun* **when the result refers to the same object as the source glvalue**. —*end footnote*]

2177. Placement `operator delete` and parameter copies

Section: 8.3.4 [expr.new] **Status:** DR **Submitter:** Richard Smith **Date:** 2015-09-30

[Accepted as a DR at the November, 2017 meeting.]

Consider the following example:

```
void *operator new(size_t n, std::string s) {
  std::string t = std::move(s);
  std::cout << "new " << t << std::endl;
  return operator new(n);
}
void operator delete(void*, std::string s) {
  std::cout << "delete " << s << std::endl;
}
struct X { X() { throw 0; } };
int main() {
  try {
    new ("longer than the small string buffer") X();
  } catch (...) {}
}
```

Current implementations print

```
new longer than the small string buffer
delete
```

because the same `std::string` object is used for both the new and delete calls. We should introduce additional copies to separate out the parameters in this case or make non-trivially-copyable parameter types ill-formed here.

Notes from the October, 2015 meeting:

CWG favored limiting the parameters of an overloaded deallocation function to trivially-copyable types.

Proposed resolution (October, 2017):

Change 8.3.4 [expr.new] paragraph 24 as follows:

If a *new-expression* calls a deallocation function, it passes the value returned from the allocation function call as the first argument of type `void*`. If a placement deallocation function is called, it is passed the same additional arguments as were passed to the placement allocation function, that is, the same arguments as those specified with the *new-placement* syntax. If the implementation is allowed to **introduce a temporary object or** make a copy of any argument as part of the call to the allocation function, it is ~~allowed to make a copy (of the same original value) as part of the call to the deallocation function or to reuse the copy made as part of the call to the allocation function. If the copy is elided in one place, it need not be elided in the other~~ **unspecified whether the same object is used in the call to both the allocation and deallocation functions**,

---

2226. Xvalues vs lvalues in conditional expressions

Section: 8.16 [expr.cond] **Status:** DR **Submitter:** Richard Smith **Date:** 2016-02-01

[Accepted at the March, 2018 (Jacksonville) meeting.]

In the following example,

```
const T a;
T b;
false ? a : std::move(b);
```

the most appropriate result would seem to be that the expression is an lvalue of type `const T` that refers to either `a` or `b`. However, because 8.16 [expr.cond] bullet 4.1 requires that the conversion bind directly to an lvalue, while `std::move(b)` is an xvalue, the result is a `const T` temporary copy-initialized from `std::move(b)`.

Proposed resolution (November, 2017)

Change 8.16 [expr.cond] bullet 4.1 as follows:

> Attempts are made to form an implicit conversion sequence from an operand expression `E1` of type `T1` to a target type related to the type `T2` of the operand expression `E2` as follows:
>
> - If `E2` is an lvalue, the target type is "lvalue reference to `T2`", subject to the constraint that in the conversion the reference must bind directly (11.6.3 [dcl.init.ref]) to ~~an lvalue~~ **a glvalue**.
> - ...

2299. `constexpr` vararg functions

Section: 10.1.5 [dcl.constexpr] **Status:** DR **Submitter:** Daveed Vandevoorde **Date:** 2016-04-11

[Accepted at the March, 2018 (Jacksonville) meeting.]

It is not clear whether a `constexpr` function can be a vararg function or not. In particular, it is unclear if `va_list` is a literal type and whether `va_start`, `va_arg`, and `va_end` produce constant expressions.

Proposed resolution (November, 2017)

1. Add a new bullet to the list in 8.20 [expr.const] paragraph 2, and update the text as follows:

> An expression `e` is a core constant expression unless the evaluation of `e`, following the rules of the abstract machine (4.6 [intro.execution]), would evaluate one of the following expressions:
>
> - ...
> - a relational (8.9 [expr.rel]) or equality (8.10 [expr.eq]) operator where the result is unspecified; ~~or~~
> - a *throw-expression* (8.17 [expr.throw])~~.~~ **; or**
> - **an invocation of the `va_arg` macro (21.10.1 [cstdarg.syn]).**
>
> If `e` satisfies the constraints of a core constant expression, but evaluation of `e` would evaluate an operation that has undefined behavior as specified in Clause 20 [library] through Clause 33 [thread] of this document, **or an invocation of the `va_start` macro (21.10.1 [cstdarg.syn]),** it is unspecified whether `e` is a core constant expression.

2059. Linkage and deduced return types

Section: 10.1.7.4 [dcl.spec.auto] **Status:** DR **Submitter:** Richard Smith **Date:** 2014-12-15

[Accepted at the March, 2018 (Jacksonville) meeting.]

Use of function return type deduction makes it possible to define functions whose return type is a type without linkage. Although 6.5 [basic.link] paragraph 8 permits such a usage if the function is defined in the same translation unit as it is used, it may be helpful to consider changing the overall rules regarding the use of types with internal or no linkage. As an example, the following example permits access to a local static variable that has not been initialized:

```
auto f() {
  static int n = 123;
  struct X { int &f() { return n; } };
  return X();
}
int &r = decltype(f())().f();
```

Notes from the February, 2016 meeting:

CWG agreed that the current rule in 6.5 [basic.link] paragraph 8 is unneeded; the ODR already prohibits use of an entity that is not defined in the current translation unit and cannot be defined in a different translation unit.

Proposed resolution (November, 2017)

Change 6.5 [basic.link] paragraph 8 as follows:

> ...~~A type without linkage shall not be used as the type of a variable or function with external linkage unless~~

- ~~the entity has C language linkage (10.5 [dcl.link]), or~~
- ~~the entity is declared within an unnamed namespace (10.3.1 [namespace.def]), or~~
- ~~the entity is not odr-used (6.2 [basic.def.odr]) or is defined in the same translation unit.~~

[*Note:* In other words, a type without linkage contains a class or enumeration that cannot be named outside its translation unit. ~~An entity with external linkage declared using such a type could not correspond to any other entity in another translation unit of the program and thus must be defined in the translation unit if it is odr-used. Also note that classes~~ **Classes** with linkage may contain members whose types do not have linkage~~, and that typedef~~**. Typedef** names are ignored in the determination of whether a type has linkage. —*end note*] [*Example:*

```
template <class T> struct B {
  void g(T) { }
  void h(T);
  friend void i(B, T) { }
};

void f() {
  struct A { int x; };  // no linkage
  A a = { 1 };
  B<A> ba;              // declares B<A>::g(A) and B<A>::h(A)
  ba.g(a);              // OK
  ba.h(a);              // error: B<A>::h(A) not defined; A cannot be named in ~~the~~ another translation unit
  i(ba, a);             // OK
}
```

—*end example*]

2081. Deduced return type in redeclaration or specialization of function template

Section: 10.1.7.4 [dcl.spec.auto] **Status:** DR **Submitter:** John Spicer **Date:** 2015-02-05

[Accepted at the March, 2018 (Jacksonville) meeting.]

10.1.7.4 [dcl.spec.auto] paragraph 13 says,

> Redeclarations or specializations of a function or function template with a declared return type that uses a placeholder type shall also use that placeholder, not a deduced type.

The inverse should also be true (a specialization cannot use a placeholder type if the template used a non-placeholder), but this is not said explicitly.

Proposed resolution (November, 2017)

Change 10.1.7.4 [dcl.spec.auto] paragraph 11 as follows:

> Redeclarations or specializations of a function or function template with a declared return type that uses a placeholder type shall also use that placeholder, not a deduced type. **Similarly, redeclarations or specializations of a function or function template with a declared return type that does not use a placeholder type shall not use a placeholder.** [*Example:*

2346. Local variables in default arguments

Section: 11.3.6 [dcl.fct.default] **Status:** DR **Submitter:** Geoffrey Romer **Date:** 2017-04-26

[Adopted as a DR as part of paper P0588R1 at the October, 2018 meeting.]

According to 11.3.6 [dcl.fct.default] paragraph 7,

> A local variable shall not appear as a potentially-evaluated expression in a default argument.

This prohibits plausible uses of `constexpr` and `static` local variables. Presumably this rule should be similar to the one in 12.4 [class.local] paragraph 1, regarding local classes, which applies to odr-use, not potential evaluation, and to variables with automatic storage duration.

2313. Redeclaration of structured binding reference variables

Section: 11.5 [dcl.struct.bind] **Status:** DR **Submitter:** Richard Smith **Date:** 2016-08-12

[Accepted as a DR at the November, 2017 meeting.]

According to the current rules for structured binding declarations, the user-defined case declares the bindings as variables of reference type. This presumably makes an example like the following valid:

```
auto [a] = std::tuple<int>(0);
extern int &&a; // ok, redeclaration, could even be in a different TU
```

This seems unreasonable, especially in light of the fact that it only works for the user-defined case and not the built-in case (where the bindings are not modeled as references).

Proposed resolution (August, 2017):

Change 11.5 [dcl.struct.bind] paragraph 3 as follows:

> Otherwise, if the *qualified-id* `std::tuple_size<E>` names a complete type, the expression `std::tuple_size<E>::value` shall be a well-formed integral constant expression and the number of elements in the *identifier-list* shall be equal to the value of that expression. The *unqualified-id* `get` is looked up in the scope of `E` by class member access lookup (6.4.5 [basic.lookup.classref]), and if that finds at least one declaration, the initializer is `e.get<i>()`. Otherwise, the initializer is `get<i>(e)` where `get` is looked up in the associated namespaces (6.4.2 [basic.lookup.argdep]). In either case, `get<i>` is interpreted as a *template-id*. [*Note:* Ordinary unqualified lookup (6.4.1 [basic.lookup.unqual]) is not performed. —*end note*] In either case, `e` is an lvalue if the type of the entity `e` is an lvalue reference and an xvalue otherwise. Given the type T_i designated by `std::tuple_element<i, E>::type`, ~~each v_i is a variable~~ **variables are introduced with unique names r_i** of type "reference to T_i" initialized with the initializer **(11.6.3 [dcl.init.ref])**, where the reference is an lvalue reference if the initializer is an lvalue and an rvalue reference otherwise**. Each v_i is the name of an lvalue of type T_i that refers to the object bound to r_i** the referenced type is T_i.

2234. Missing rules for *simple-template-id* as *class-name*

Section: 12 [class] **Status:** DR **Submitter:** Richard Smith **Date:** 2016-02-25

[Accepted at the March, 2018 (Jacksonville) meeting.]

There does not seem to be a rule that prohibits an example like:

```
template<typename T> struct X;
struct X<int> {
};
```

Proposed resolution (November, 2017)

Change 12 [class] paragraph 1 as follows:

> ...**A class declaration where the *class-name* in the *class-head-name* is a *simple-template-id* shall be an explicit specialization (17.8.3 [temp.expl.spec]) or a partial specialization (17.6.5 [temp.class.spec]).** A *class-specifier* whose *class-head* omits the *class-head-name* defines an unnamed class. [*Note:* An unnamed class thus can't be final. —*end note*]

1983. Inappropriate use of *virt-specifier*

Section: 12.2 [class.mem] **Status:** DR **Submitter:** Richard Smith **Date:** 2014-08-11

[Accepted at the March, 2018 (Jacksonville) meeting.]

The restriction in 12.2 [class.mem] paragraph 8 that a *virt-specifier* may appear only in the declaration of a virtual function is insufficient to rule out examples like the following:

```
struct A { virtual void f(); };
struct B { friend void A::f() final; };

template<typename T> struct C { virtual void f() {} };
template void C<int>::f() final;
template<> void C<char>::f() final;
```

One possibility might be to require that a *virt-specifier* appear only on the first declaration of a function.

Proposed resolution (November, 2017)

Change 12.2 [class.mem] paragraph 13 as follows:

> A *virt-specifier-seq* shall contain at most one of each *virt-specifier*. A *virt-specifier-seq* shall appear only in the **first** declaration of a virtual member function (13.3 [class.virtual]).

2229. Volatile unnamed bit-fields

Section: 12.2.4 [class.bit] **Status:** DR **Submitter:** David Majnemer **Date:** 2016-02-08

[Accepted at the March, 2018 (Jacksonville) meeting.]

According to 12.2.4 [class.bit] paragraph 2, unnamed bit-fields are not members, but there does not appear to be a prohibition against their being declared `volatile`. Is this intended?

Proposed resolution (November, 2017)

Change 12.2.4 [class.bit] paragraph 2 as follows:

> A declaration for a bit-field that omits the *identifier* declares an *unnamed bit-field*. Unnamed bit-fields are not members and cannot be initialized. **An unnamed bit-field shall not be declared with a cv-qualified type.** [*Note:* An unnamed bit-field is useful for padding...

2227. Destructor access and default member initializers

Section: 15.6.2 [class.base.init] **Status:** DR **Submitter:** Richard Smith **Date:** 2016-02-01

[Accepted at the March, 2018 (Jacksonville) meeting.]

There is implementation divergence on the validity of the following:

```
class X { ~X(); };
struct Y { X x = {}; };
```

Should `X`'s destructor be potentially invoked by this attempt to initialize an `X` object? Or,

```
auto *y = new Y {};
```

No constructor for `Y` is used, because this is aggregate initialization, and a destructor for `X` is not strictly necessary as there is no later initialization that might throw, but in the corresponding default constructor case we do require that the destructor be valid.

Perhaps the most consistent answer is that the default member initializer should not potentially invoke the destructor unless it's used (for symmetry with default arguments), but that aggregate initialization should potentially invoke the destructors of all subobjects (including the final one - exceptions could theoretically be thrown between the completion of the construction of the final aggregate element and the notional completion of the construction of the aggregate itself.

Proposed resolution (November, 2017)

1. Add the following as a new paragraph following 11.6.1 [dcl.init.aggr] paragraph 7:

> An aggregate that is a class can also be initialized with a single expression not enclosed in braces, as described in 11.6 [dcl.init].
>
> **The destructor for each element of class type is potentially invoked (15.4 [class.dtor]) from the context where the aggregate initialization occurs. [*Note:* This provision ensures that destructors can be called for fully-constructed subobjects in case an exception is thrown (18.2 [except.ctor]). —*end note*]**

2. Change 15.4 [class.dtor] paragraph 12 as follows:

> ...A destructor is *potentially invoked* if it is invoked or as specified in 8.3.4 [expr.new], **11.6.1 [dcl.init.aggr],** 15.6.2 [class.base.init], and 18.1 [except.throw]. A program is ill-formed if a destructor that is potentially invoked is deleted or not accessible from the context of the invocation.

2315. What is the "corresponding special member" of a variant member?

Section: 15.8 [class.copy] **Status:** DR **Submitter:** Richard Smith **Date:** 2016-08-15

[Accepted as a DR at the November, 2017 meeting.]

According to 15.8.1 [class.copy.ctor] bullet 10.1,

> A defaulted copy/move constructor for a class `X` is defined as deleted (11.4.3 [dcl.fct.def.delete]) if `X` has:
>
> - a variant member with a non-trivial corresponding constructor and `X` is a union-like class,

However, it is not clear from this specification how to handle an example like:

```
struct A {
  A();
  A(const A&);
};
union B {
  A a;
};
```

since there is no corresponding special member in `A`.

Proposed resolution (August, 2017):

Change 15.8.1 [class.copy.ctor] paragraph 10 as follows:

An implicitly-declared copy/move constructor is an inline public member of its class. A defaulted copy/move constructor for a class `X` is defined as deleted (11.4.3 [dcl.fct.def.delete]) if `X` has:

- ~~a variant member with a non-trivial corresponding constructor and `X` is a union-like class,~~
- a potentially constructed subobject type `M` (or array thereof) that cannot be copied/moved because overload resolution (16.3 [over.match]), as applied to find `M`'s corresponding constructor, results in an ambiguity or a function that is deleted or inaccessible from the defaulted constructor,
- **a variant member whose corresponding constructor as selected by overload resolution is non-trivial,**
- any potentially constructed subobject of a type with a destructor that is deleted or inaccessible from the defaulted constructor, or,
- for the copy constructor, a non-static data member of rvalue reference type.

1862. Determining "corresponding members" for friendship

Section: 17.6.4 [temp.friend] **Status:** DR **Submitter:** Richard Smith **Date:** 2014-02-13

[Accepted as a DR at the November, 2017 meeting.]

During the discussion of issue 1804, it was noted that the process of determining whether a member of an explicit or partial specialization corresponds to a member of the primary template is not well specified. In particular, it should be clarified that the primary template should not be instantiated during this process; instead, the template arguments from the specialization should simply be substituted into the member declaration.

Proposed resolution (October, 2017):

Change 17.6.4 [temp.friend] paragraph 4 as follows:

A **template friend declaration may declare a** member of a ~~class template may be declared~~ **dependent type** to be a friend ~~of a non-template class~~. **The friend declaration shall declare a function or specify a type with an *elaborated-type-specifier*, in either case with a *nested-name-specifier* ending with a *simple-template-id*, *C*, whose *template-name* names a class template. The template parameters of the template friend declaration shall be deducible from *C* (17.9.2.5 [temp.deduct.type]).** In this case, ~~the corresponding member of every specialization of the primary class template and class template partial specializations thereof~~ **a member of a specialization *S* of the class template** is a friend of the class granting friendship~~. For explicit specializations and specializations of partial specializations, the corresponding member is the member (if any) that has the same name, kind (type, function, class template, or function template), template parameters, and signature as the member of the class template instantiation that would otherwise have been generated~~ **if deduction of the template parameters of *C* from *S* succeeds, and substituting the deduced template arguments into the friend declaration produces a declaration that would be a valid redeclaration of the member of the specialization**. [*Example:*

```
template<class T> struct A {
  struct B { };
  void f();
  struct D {
    void g();
  };
   T h();
   template<T U> T i();
};
template<> struct A<int> {
  struct B { };
  int f();
  struct D {
    void g();
  };
   template<int U> int i();
};
template<> struct A<float*> {
   int *h();
};
class C {
  template<class T> friend struct A<T>::B;      // grants friendship to A<int>::B even though
                                                // it is not a specialization of A<T>::B
  template<class T> friend void A<T>::f();      // does not grant friendship to A<int>::f()
                                                // because its return type does not match
  template<class T> friend void A<T>::D::g();   // ~~does not grant friendship to A<int>::D::g()~~
                                                // ~~because A<int>::D is not a specialization of A<T>::D~~ ill-formed, A<T>::D does not end with a simple-te
   template<class T> friend int *A<T*>::h();    // grants friendship to A<int*>::h() and A<float*>::h()
   template<class T> template<T U>              // grants friendship to instantiations of A<T>::i() and to A<int>::i()
     friend T A<T>::i();                        // and thereby to all specializations of those function templates
};
```

—*end example*]

1271. Imprecise wording regarding dependent types

Section: 17.7 [temp.res] **Status:** DR **Submitter:** Daveed Vandevoorde **Date:** 2011-03-24

[Accepted at the March, 2018 (Jacksonville) meeting as part of paper P0634R3.]

According to 17.7 [temp.res] paragraph 3,

> When a *qualified-id* is intended to refer to a type that is not a member of the current instantiation (17.7.2.1 [temp.dep.type]) and its *nested-name-specifier* refers to a dependent type, it shall be prefixed by the keyword `typename`, forming a *typename-specifier*. If the *qualified-id* in a *typename-specifier* does not denote a type, the program is ill- formed.

The intent of the programmer cannot form the basis for a compiler determining whether to issue a diagnostic or not.

Suggested resolution:

> Let `N` be a *qualified-id* with a *nested-name-specifier* that denotes a dependent type. If `N` is not prefixed by the keyword `typename`, `N` shall refer to a member of the current instantiation or it shall not refer to a type.
>
> *typename-specifier:*
> `typename` *nested-name-specifier identifier*
> `typename` *nested-name-specifier* `template`$_{opt}$ *simple-template-id*
>
> If the *qualified-id* in a *typename-specifier* does not denote a type, the program is ill-formed.

(See also issues 590 and 591.)

Notes from the November, 2016 meeting:

The resolution for this issue should describe the type to which a *typename-specifier* refers, effectively the type named by the corresponding *simple-type-specifier* with `typename` removed.

Notes from the November, 2017 meeting:

This topic is addressed in paper P0634.

2307. Unclear definition of "equivalent to a nontype template parameter"

Section: 17.7.2.1 [temp.dep.type] **Status:** DR **Submitter:** Richard Smith **Date:** 2016-07-21

[Accepted as a DR at the November, 2017 meeting.]

The description of whether a template argument is equivalent to a template parameter in 17.7.2.1 [temp.dep.type] paragraph 3 is unclear as it applies to non-type template parameters:

> A template argument that is equivalent to a template parameter (i.e., has the same constant value or the same type as the template parameter) can be used in place of that template parameter in a reference to the current instantiation. In the case of a non-type template argument, the argument must have been given the value of the template parameter and not an expression in which the template parameter appears as a subexpression.

For example:

```
template<int N> struct A {
  typedef int T[N];
  static const int AlsoN = N;
  A<AlsoN>::T s; // #0, clearly supposed to be OK
  static const char K = N;
  A<K>::T t;     // #1, OK?
  static const long L = N;
  A<L>::T u;     // #2, OK?
  A<(N)>::T v;   // #3, OK?
  static const int M = (N);
  A<M>::T w;     // #4, OK?
};
```

#1 narrows the template argument. This obviously should not be the injected-class-name, because `A<257>::T` may well be `int[1]` not `int[257]`. However, the wording above seems to treat it as the injected-class-name.

#2 is questionable: there is potentially a narrowing conversion here, but it doesn't actually narrow any values for the original template parameter.

#3 is hard to decipher. On the one hand, this is an expression involving the template parameter. On the other hand, a parenthesized expression is specified as being equivalent to its contained expression.

#4 should presumably go the same way that #3 does.

Proposed resolution (August, 2017):

Change 17.7.2.1 [temp.dep.type] paragraph 3 as follows:

A template argument that is equivalent to a template parameter ~~(i.e., has the same constant value or the same type as the template parameter)~~ can be used in place of that template parameter in a reference to the current instantiation. **For a template *type-parameter*, a template argument is equivalent to a template parameter if it denotes the same type. For a non-type template parameter, a template argument is equivalent to a template parameter if it is an *identifier* that names a variable that is equivalent to the template parameter. A variable is equivalent to a template parameter if**

- **it has the same type as the template parameter (ignoring cv-qualification) and**
- **its initializer consists of a single *identifier* that names the template parameter or, recursively, such a variable.**

[*Note:* Using a parenthesized variable name breaks the equivalence. *—end note*] ~~In the case of a non-type template argument, the argument must have been given the value of the template parameter and not an expression in which the template parameter appears as a subexpression.~~ [*Example:*

```
template <class T> class A {
  A* p1;                    // A is the current instantiation
  A<T>* p2;                 // A<T> is the current instantiation
  A<T*> p3;                 // A<T*> is not the current instantiation
  ::A<T>* p4;               // ::A<T> is the current instantiation
  class B {
    B* p1;                  // B is the current instantiation
    A<T>::B* p2;            // A<T>::B is the current instantiation
    typename A<T*>::B* p3; // A<T*>::B is not the current instantiation
  };
};

template <class T> class A<T*> {
  A<T*>* p1;                // A<T*> is the current instantiation
  A<T>* p2;                 // A<T> is not the current instantiation
};

template <class T1, class T2, int I> struct B {
  B<T1, T2, I>* b1;         // refers to the current instantiation
  B<T2, T1, I>* b2;         // not the current instantiation
  typedef T1 my_T1;
  static const int my_I = I;
  static const int my_I2 = I+0;
  static const int my_I3 = my_I;
   static const long my_I4 = I;
   static const int my_I5 = (I);
  B<my_T1, T2, my_I>* b3;    // refers to the current instantiation
  B<my_T1, T2, my_I2>* b4;   // not the current instantiation
  B<my_T1, T2, my_I3>* b5;   // refers to the current instantiation
   B<my_T1, T2, my_I4>* b6;  // not the current instantiation
   B<my_T1, T2, my_I5>* b7;  // not the current instantiation
};
```

—end example]

Additional note (November, 2017):

It was observed that the proposed resolution does not address partial specializations, which also depend on the definition of equivalence. For example:

```
template <typename T, unsigned N> struct A;
template <typename T> struct A<T, 42u> {
  typedef int Ty;
  static const unsigned num = 42u;
  static_assert(!A<T, num>::Ty(), "");
};
A<int, 42u> a; // GCC, MSVC, ICC accepts; Clang rejects
```

The issue is being returned to "review" status in order to consider these additional questions.

Notes from the November, 2017 (Albuquerque) meeting:

CWG decided to proceed with this resolution and deal with partial specialization in a separate issue.

2255. Instantiated static data member templates

Section: 17.8 [temp.spec] **Status:** DR **Submitter:** Mike Miller **Date:** 2016-03-29

[Accepted at the March, 2018 (Jacksonville) meeting.]

The current wording does not state that a specialization of a static data member template (17 [temp] paragraph 1) is a static data member, which leaves the status of an example like the following unclear (since 8.2.5 [expr.ref] bullet 4.1 is phrased in terms of static data members):

```
template <class T> struct A {
  template <class U> static const U x = 1;
  static const int y = 2;
};

int main() {
  A<int> a;
```

```
  int y = a.y;          // OK
  int x = a.x<int>;     // ???
}
```

Proposed resolution (November, 2017)

Change 17.8 [temp.spec] paragraph 2 as follows:

> A function instantiated from a function template is called an instantiated function. A class instantiated from a class template is called an instantiated class. A member function, a member class, a member enumeration, or a static data member of a class template instantiated from the member definition of the class template is called, respectively, an instantiated member function, member class, member enumeration, or static data member. A member function instantiated from a member function template is called an instantiated member function. A member class instantiated from a member class template is called an instantiated member class. **A variable instantiated from a variable template is called an instantiated variable. A static data member instantiated from a static data member template is called an instantiated static data member.**

2305. Explicit instantiation of constexpr or inline variable template

Section: 17.8.2 [temp.explicit] **Status:** DR **Submitter:** John Spicer **Date:** 2016-07-14

[Accepted as a DR at the November, 2017 meeting.]

According to 17.8.2 [temp.explicit] paragraph 1,

> An explicit instantiation of a function template or member function of a class template shall not use the `inline` or `constexpr` specifiers.

Should this apply to explicit specializations of variable templates as well?

(See also issues 1704 and 1728).

Proposed resolution (August, 2017):

Change 17.8.2 [temp.explicit] paragraph 1 as follows:

> A class, function, variable, or member template specialization can be explicitly instantiated from its template. A member function, member class or static data member of a class template can be explicitly instantiated from the member definition associated with its class template. An explicit instantiation of a function template ~~or~~**,** member function of a class template**, or variable template** shall not use the `inline` or `constexpr` specifiers.

2260. Explicit specializations of deleted member functions

Section: 17.8.3 [temp.expl.spec] **Status:** DR **Submitter:** Richard Smith **Date:** 2016-04-17

[Accepted at the March, 2018 (Jacksonville) meeting.]

Although the Standard allows for explicitly specializing a deleted function template, member function of a class template, or member function template with a non-deleted definition, this seems to be problematic for non-template member functions of class templates. For example:

```
template<typename T> struct A {
  A(const A&) = delete;
  A(A&&) = default;
};
static_assert(is_trivially_copyable(A<int>));
template<> struct A<int>::A(const A&) { /* ... */ }
static_assert(is_trivially_copyable(A<int>));
template<typename T> struct B {
  virtual void f() = delete;
};
struct C : B<int> { void f() override = delete; }; // ok, overriding deleted with deleted
template<> void B<int>::f() {} // would make C retroactively ill-formed?
```

Notes from the December, 2016 teleconference:

`=delete` definitions of member functions should be instantiated when instantiating a class template. That would make the example an ill-formed redefinition.

Proposed resolution (November, 2017)

Change 17.8.1 [temp.inst] paragraph 2, breaking the running text into bullets, as follows:

> The implicit instantiation of a class template specialization causes
>
> - the implicit instantiation of the declarations, but not of the definitions, ~~default arguments, or *noexcept-specifiers*~~ of the **non-deleted** class member functions, member classes, scoped member enumerations, static data members, member

templates, and friends; and

- ~~it causes~~ the implicit instantiation of the definitions of **deleted member functions,** unscoped member enumerations**,** and member anonymous unions.

The implicit instantiation of a class template specialization does not cause the implicit instantiation of default arguments or *noexcept-specifier*s of the class member functions. [*Example:*

```
template<class T>
struct C {
  void f() { T x; }
  void g() = delete;
};
C<void> c;                          // OK, definition of C<void>::f is not instantiated at this point
template<> void C<int>::g() { }     // error: redefinition of C<int>::g
```

***—end example*]** However, for the purpose of determining whether an instantiated redeclaration is valid according to 6.2 [basic.def.odr] and 12.2 [class.mem], a declaration that corresponds to a definition in the template is considered to be a definition. [*Example:*

2088. Late tiebreakers in partial ordering

Section: 17.9.2.4 [temp.deduct.partial] **Status:** DR **Submitter:** Richard Smith **Date:** 2015-02-19

[Accepted at the March, 2018 (Jacksonville) meeting.]

The late tiebreakers for lvalue-vs-rvalue references and cv-qualification in 17.9.2.4 [temp.deduct.partial] paragraph 9 are applied

> If, for a given type, deduction succeeds in both directions (i.e., the types are identical after the transformations above) and both `P` and `A` were reference types (before being replaced with the type referred to above):

However, this is based on a false assumption. For example,

```
template <typename T> struct A {
  struct typeA { };
  struct typeB { };
  using convTyA = T (*const &&)(typename A<T>::typeA);
  using convTyB = T (*const &)(typename A<T>::typeB);
  operator convTyA();
  operator convTyB();
};

template <typename T> void foo(T (*const &&)(typename A<T>::typeA));
template <typename T> int foo(T (*const &)(typename A<T>::typeB));

int main() {
  return foo<int>(A<int>());
}
```

(see also issues 1847 and 1157.). We need to decide whether the rule is "deduction succeeds in both directions" or "the types are identical." The latter seems more reasonable.

Proposed resolution (November, 2017)

Change 17.9.2.4 [temp.deduct.partial] paragraph 9 as follows:

> If, for a given type, ~~deduction succeeds in both directions (i.e.,~~ the types are identical after the transformations above~~)~~ and both `P` and `A` were reference types (before being replaced with the type referred to above):
>
> - if the type from the argument template was an lvalue reference and the type from the parameter template was not, the parameter type is not considered to be at least as specialized as the argument type; otherwise,
> - if the type from the argument template is more cv-qualified than the type from the parameter template (as described above), the parameter type is not considered to be at least as specialized as the argument type.

2235. Partial ordering and non-dependent types

Section: 17.9.2.4 [temp.deduct.partial] **Status:** DR **Submitter:** Richard Smith **Date:** 2016-02-25

[Accepted at the March, 2018 (Jacksonville) meeting.]

Paragraph 12 of 17.9.2.4 [temp.deduct.partial] contains the following example:

```
template <class T> T f(int);        // #1
template <class T, class U> T f(U); // #2
void g() {
  f<int>(1);                        // calls #1
}
```

However, paragraph 4 states,

> If a particular `P` contains no *template-parameter*s that participate in template argument deduction, that `P` is not used to determine the ordering.

Thus, we ignore the `P=int, A=U` case and deduction succeeds for the `P=U, A=int` case, so both templates are at least as specialized as each other. And consider:

```
template <class... T> struct V {};
template <class... Ts, class... Us> void Foo(V<Ts...>, V<Us&...>) {} // #3
template <class... Us> void Foo(V<>, V<Us&...>) {}                   // #4
void h() {
  Foo(V<>(), V<>());
}
```

The intent is that this should call #4; that template clearly ought to be more specialized.

Proposed resolution (November, 2017)

1. Change 17.9.2.4 [temp.deduct.partial] paragraph 4 as follows:

   :

   > Each type nominated above from the parameter template and the corresponding type from the argument template are used as the types of `P` and `A`. ~~If a particular `P` contains no *template-parameter*s that participate in template argument deduction, that `P` is not used to determine the ordering.~~

2. Change 17.9.2.5 [temp.deduct.type] paragraph 4 as follows:

   > ...If a template parameter is used only in non-deduced contexts and is not explicitly specified, template argument deduction fails. [*Note:* Under 17.9.2.1 [temp.deduct.call] ~~and 17.9.2.4 [temp.deduct.partial]~~, if `P` contains no *template-parameter*s that appear in deduced contexts, no deduction is done, so `P` and `A` need not have the same form. —*end note*]

2092. Deduction failure and overload resolution

Section: 17.9.3 [temp.over] **Status:** DR **Submitter:** Fedor Sergeev **Date:** 2015-03-06

[Accepted at the March, 2018 (Jacksonville) meeting.]

Given an example like

```
template <class T = int> void foo(T*);

void test()
{
  foo(0);   // #1 valid?
  foo<>(0);  // #2 valid?
}
```

most/all implementations reject this code. However, the wording of the Standard only invokes 17.9.3 [temp.over] ("Overload resolution") in cases where there is more than one function or function template, which is not the case here. The current wording would appear to make this well-formed because of the application of 17.9.1 [temp.arg.explicit] paragraph 2. Perhaps overload resolution should apply even when there is a single function template?

Notes from the May, 2015 meeting:

This issue is mostly a duplicate of issue 1582. However, CWG felt that it should be clarified that overload resolution applies in all cases, not just when templates are overloaded, so the issue is being left open to deal with that aspect.

Proposed resolution (November, 2017)

1. Change 8.2.2 [expr.call] paragraph 1, splitting it into three paragraphs, as follows:

   > A function call is a postfix expression followed by parentheses containing a possibly empty, comma-separated list of *initializer-clause*s which constitute the arguments to the function. The postfix expression shall have function type or function pointer type. For a call to a non-member function or to a static member function, the postfix expression shall be either an lvalue that refers to a function (in which case the function-to-pointer standard conversion (7.3 [conv.func]) is suppressed on the postfix expression), or it shall have function pointer type. ~~Calling a function through an expression whose function type is different from the function type of the called function's definition results in undefined behavior (10.5 [dcl.link]).~~
   >
   > For a call to a non-static member function, the postfix expression shall be an implicit (12.2.2 [class.mfct.non-static], 12.2.3 [class.static]) or explicit class member access (8.2.5 [expr.ref]) whose *id-expression* is a function member name, or a pointer-to-member expression (8.5 [expr.mptr.oper]) selecting a function member; the call is as a member of the class object referred to by the object expression. In the case of an implicit class member access, the implied object is the one pointed to by `this`. [*Note:* A member function call of the form `f()` is interpreted as `(*this).f()` (see 12.2.2 [class.mfct.non-static]). —*end note*]

If a function or member function name is used, ~~the name can be overloaded (Clause 16 [over]), in which case~~ the appropriate function ~~shall be selected~~ **and the validity of the call are determined** according to the rules in 16.3 [over.match]. If the selected function is non-virtual, or if the *id-expression* in the class member access expression is a qualified-id, that function is called. Otherwise, its final overrider (13.3 [class.virtual]) in the dynamic type of the object expression is called; such a call is referred to as a *virtual function call*. [*Note:* The dynamic type is the type of the object referred to by the current value of the object expression. 15.7 [class.cdtor] describes the behavior of virtual function calls when the object expression refers to an object under construction or destruction. —*end note*]

2. Add the following to 8.2.2 [expr.call] as a new paragraph before the existing paragraph 4:

> **Calling a function through an expression whose function type is different from the function type of the called function's definition results in undefined behavior (10.5 [dcl.link]).**
>
> When a function is called, each parameter (11.3.5 [dcl.fct]) shall be initialized...

3. Change 16 [over] paragraph 2 as follows:

> When ~~an overloaded~~ **a** function name is used in a call, which ~~overloaded~~ function declaration is being referenced ~~is~~ **and the validity of the call are** determined by comparing the types of the arguments at the point of use with the types of the parameters in the ~~overloaded~~ declarations that are visible at the point of use. This function selection process is called *overload resolution*...

4. Change 17.9.3 [temp.over] paragraph 1 as follows:

> ~~A function template can be overloaded either by (non-template) functions of its name or by (other) function templates of the same name.~~ When a call to ~~that~~ **the** name **of a function or function template** is written (explicitly, or implicitly using the operator notation), template argument deduction...

This resolution also resolves issue 2241.

Issues with "accepted" Status

2323. Expunge POD

Section: 6.9 [basic.types] **Status:** accepted **Submitter:** US **Date:** 2017-02-27

[Adopted at the October, 2017 meeting as paper P0767R1.]

P0488R0 comment US 101

The term POD no longer serves a purpose in the standard, it is merely defined, and restrictions apply for when a few other types preserve this vestigial property. The `is_pod` trait should be deprecated, moving the definition of a POD type alongside the trait in Annex D [depr], and any remaining wording referring to POD should be struck, or revised to clearly state intent (usually triviality) without mentioning PODs.

2237. Can a *template-id* name a constructor?

Section: 15.1 [class.ctor] **Status:** accepted **Submitter:** Faisal Vali **Date:** 2016-03-02

[Accepted at the March, 2018 (Jacksonville) meeting.]

Bullet 1.2 of 15.1 [class.ctor], describing declarator forms that are considered to declare a constructor, says:

> ...and the *id-expression* has one of the following forms:
>
> - ...
> - in a *member-declaration* that belongs to the *member-specification* of a class template but is not a friend declaration, the *id-expression* is a *class-name* that names the current instantiation (17.7.2.1 [temp.dep.type]) of the immediately-enclosing class template; or
> - ...

The term *class-name* includes *simple-template-id*. It is not clear that allowing a constructor declaration of the form

```
template<class T> struct X {
  X<T>(T); // constructor
};
```

is useful or helpful.

Proposed resolution (November, 2017)

1. Change 15.1 [class.ctor] paragraph 1 as follows:

   ...and the *id-expression* has one of the following forms:

   - in a *member-declaration* that belongs to the *member-specification* of a class **or class template** but is not a friend declaration (14.3 [class.friend]), the *id-expression* is the injected-class-name (Clause 12 [class]) of the immediately-enclosing ~~class;~~ **entity or**
   - ~~in a *member-declaration* that belongs to the *member-specification* of a class template but is not a friend declaration, the *id-expression* is a *class-name* that names the current instantiation (17.7.2.1 [temp.dep.type]) of the immediately-enclosing class template; or~~
   - in a declaration at namespace scope or in a friend declaration, the *id-expression* is a *qualified-id* that names a constructor (6.4.3.1 [class.qual]).

2. Change 15.4 [class.dtor] paragraph 1 as follows:

   ...and the *id-expression* has one of the following forms:

   - in a *member-declaration* that belongs to the *member-specification* of a class **or class template** but is not a friend declaration (14.3 [class.friend]), the *id-expression* is `~`*class-name* and the *class-name* is the injected-class-name (Clause 12 [class]) of the immediately-enclosing ~~class;~~ **entity or**
   - ~~in a *member-declaration* that belongs to the *member-specification* of a class template but is not a friend declaration, the *id-expression* is `~`*class-name* and the *class-name* names the current instantiation (17.7.2.1 [temp.dep.type]) of the immediately-enclosing class template; or~~
   - in a declaration at namespace scope or in a friend declaration, the *id-expression* is *nested-name-specifier* `~`*class-name* and the *class-name* names the same class as the *nested-name-specifier*.

3. Add the following as a new paragraph in C.5 [diff.cpp17]:

   C.5.x Clause 15: Special member functions [diff.cpp17.special]

   Affected subclauses: 15.1 [class.ctor], 15.4 [class.dtor]
   Change: A *simple-template-id* is no longer valid as the *declarator-id* of a constructor or destructor.
   Rationale: Remove potentially error-prone option for redundancy.
   Effect on original feature: Valid C++ 2017 code may fail to compile.

```
template<class T>
struct A {
  A<T>();   // error: simple-template-id not allowed for constructor
  A(int);   // OK, injected-class-name used
  ~A<T>(); // error: simple-template-id not allowed for destructor
};
```

(Note that this resolution is a change for C++20, NOT a defect report against C++17 and earlier versions.)

Issues with "DRWP" Status

1836. Use of class type being defined in *trailing-return-type*

Section: _N4567_.5.1.1 [expr.prim.general] **Status:** DRWP **Submitter:** Mike Miller **Date:** 2014-01-17

[Voted into the WP at the July, 2017 meeting.]

According to _N4567_.5.1.1 [expr.prim.general] paragraph 3,

> Unlike the object expression in other contexts, `*this` is not required to be of complete type for purposes of class member access (8.2.5 [expr.ref]) outside the member function body.

Is this special treatment of member access expressions intended to apply only to `*this`, or does it apply to other ways of specifying the class being defined in the object expression? For example,

```
struct S {
  int i;
  auto f1() -> decltype((*this).i);      // okay
  auto f2(S& This) -> decltype(This.i); // okay?
  auto f3() -> decltype(((S*)0)->i);     // okay?
};
```

There is implementation divergence on this question.

If the intent is to allow object expressions other than `*this` to have the current class type, this specification should be moved from _N4567_.5.1.1 [expr.prim.general] to 8.2.5 [expr.ref] paragraph 2, which is where the general requirement for complete object expression types is found.

On a related point, the note immediately following the above-cited passage is not quite correct:

> [*Note:* only class members declared prior to the declaration are visible. —*end note*]

This does not apply when the member is a "member of an unknown specialization," per 17.7.2.1 [temp.dep.type] paragraph 5 bullet 3 sub-bullet 1; for example,

```
template<typename T> struct S : T {
  auto f() -> decltype(this->x);
};
```

Here `x` is presumed to be a member of the dependent base `T` and is not "declared prior to the declaration" that refers to it.

Proposed resolution (May, 2017):

1. Change 8.1.2 [expr.prim.this] paragraph 2 as follows:

> ~~Unlike the object expression in other contexts, `*this` is not required to be of complete type for purposes of class member access (8.2.5 [expr.ref]) outside the member function body. [*Note:* Only class members declared prior to the declaration are visible. —*end note*]~~ **[*Note:* In a *trailing-return-type*, the class being defined is not required to be complete for purposes of class member access (8.2.5 [expr.ref]). Class members declared later are not visible.** [*Example:*
>
> ```
> struct A {
>   char g();
>   template<class T> auto f(T t) -> decltype(t + g())
>   { return t + g(); }
> };
> template auto A::f(int t) -> decltype(t + g());
> ```
>
> —*end example*] ***—end note*]**

2. Change 8.2.5 [expr.ref] paragraph 2 as follows, splitting the paragraph into two paragraphs as indicated:

> For the first option (dot) the first expression shall be a glvalue having ~~complete~~ class type. For the second option (arrow) the first expression shall be a prvalue having pointer to ~~complete~~ class type. **In both cases, the class type shall be complete unless the class member access appears in the definition of that class. [*Note:* If the class is incomplete, lookup in the complete class type is required to refer to the same declaration (6.3.7 [basic.scope.class]). —*end note*]**
>
> The expression `E1->E2` is converted to the equivalent form `(*(E1)).E2`; the remainder...

2287. Pointer-interconvertibility in non-standard-layout unions

Section: 6.9.2 [basic.compound] **Status:** DRWP **Submitter:** Richard Smith **Date:** 2016-07-06

[Voted into the WP at the July, 2017 meeting.]

According to 12.3 [class.union] paragraph 2,

> [*Note:* A union object and its non-static data members are pointer-interconvertible (6.9.2 [basic.compound], 8.2.9 [expr.static.cast]). As a consequence, all non-static data members of a union object have the same address. —*end note*]

However, the normative wording now only requires this for standard-layout unions.

Proposed resolution (April, 2017):

Change 6.9.2 [basic.compound] bullet 4.2 as follows:

> Two objects *a* and *b* are *pointer-interconvertible* if:
>
> - they are the same object, or
> - one is a ~~standard-layout~~ union object and the other is a non-static data member of that object (12.3 [class.union]), or
> - ...

1076. Value categories and lvalue temporaries

Section: 6.10 [basic.lval] **Status:** DRWP **Submitter:** Daniel Krügler **Date:** 2010-06-10

[Voted into the WP at the July, 2017 meeting as document P0727R0.]

The taxonomy of value categories in 6.10 [basic.lval] classifies temporaries as prvalues. However, some temporaries are explicitly referred to as lvalues (cf 18.1 [except.throw] paragraph 3).

Proposed resolution (March, 2017):

This issue is resolved by the resolution of issue 1299.

---

943. Is `T()` a temporary?

Section: 8.2.3 [expr.type.conv] **Status:** DRWP **Submitter:** Miller **Date:** 14 July, 2009

[Voted into the WP at the July, 2017 meeting as document P0727R0.]

According to 8.2.3 [expr.type.conv] paragraphs 1 and 3 (stated directly or by reference to another section of the Standard), all the following expressions create temporaries:

```
T(1)
T(1, 2)
T{1}
T{}
```

However, paragraph 2 says,

> The expression `T()`, where `T` is a *simple-type-specifier* or *typename-specifier* for a non-array complete effective object type or the (possibly cv-qualified) `void` type, creates an rvalue of the specified type, which is value-initialized (11.6 [dcl.init]; no initialization is done for the `void()` case).

This does *not* say that the result is a temporary, which means that the lifetime of the result is not specified by 15.2 [class.temporary]. Presumably this is just an oversight.

Notes from the October, 2009 meeting:

The specification in 8.2.3 [expr.type.conv] is in error, not because it fails to state that `T()` is a temporary but because it requires a temporary for the other cases with fewer than two operands. The case where `T` is a class type is covered by 15.2 [class.temporary] paragraph 1 ("a conversion that creates an rvalue"), and a temporary should *not* be created when `T` is not a class type.

Proposed resolution (March, 2017):

This issue is resolved by the resolution of issue 1299.

1523. Point of declaration in range-based `for`

Section: 9.5.4 [stmt.ranged] **Status:** DRWP **Submitter:** John Spicer **Date:** 2012-07-16

[Voted into the WP at the July, 2017 meeting.]

According to the general rule for declarations in 6.3.2 [basic.scope.pdecl] paragraph 1,

> The *point of declaration* for a name is immediately after its complete declarator (Clause 11 [dcl.decl]) and before its *initializer* (if any), except as noted below.

However, the rewritten expansion of the range-based `for` statement in 9.5.4 [stmt.ranged] paragraph 1 contradicts this general rule, so that the index variable is not visible in the *range-init*:

```
for (int i : {i}) ;   // error: i not in scope
```

(See also issue 1498 for another question regarding the rewritten form of the range-based `for`.)

Notes from the October, 2012 meeting:

EWG is discussing issue 900 and the outcome of that discussion should be taken into consideration in addressing this issue.

Notes from the April, 2013 meeting:

The approach favored by CWG for resolving this issue is to change the point of declaration of the variable in the *for-range-declaration* to be after the `)`.

Proposed resolution (May, 2017):

Add the following as a new pareagraph following 6.3.2 [basic.scope.pdecl] paragraph 9:

> The point of declaration for a function-local predefined variable (11.4 [dcl.fct.def]) is immediately before the *function-body* of a function definition.

The point of declaration for the variable or the structured bindings declared in the *for-range-declaration* of a range-based `for` statement (9.5.4 [stmt.ranged]) is immediately after the *for-range-initializer*.

The point of declaration for a template parameter...

2253. Unnamed bit-fields and zero-initialization

Section: 12.2.4 [class.bit] **Status:** DRWP **Submitter:** Aaron Ballman **Date:** 2016-03-23

[Voted into the WP at the July, 2017 meeting.]

The current wording of the Standard does not clearly state that zero-initialization applies to unnamed bit-fields.

Notes from the December, 2016 teleconference:

The consensus was that unnamed bit-fields do constitute padding; more generally, padding should be normatively defined, along the lines suggested in 12.2.4 [class.bit] paragraphs 1-2.

Proposed resolution (March, 2017):

1. Change 6.9 [basic.types] paragraph 4 as follows:

> The *object representation* of an object of type `T` is the sequence of *N* `unsigned char` objects taken up by the object of type `T`, where *N* equals `sizeof(T)`. The *value representation* of an object is the set of bits that hold the value of type `T`. **Bits in the object representation that are not part of the value representation are *padding bits*.** For trivially copyable types, the value representation is a set of bits in the object representation that determines a *value*, which is one discrete element of an implementation-defined set of values.[44]

2. Change 11.6 [dcl.init] paragraph 6 as follows:

> To *zero-initialize* an object or reference of type `T` means:
>
> - if `T` is a scalar type (6.9 [basic.types]), the object is initialized to the value obtained by converting the integer literal `0` (zero) to `T`;[103]
> - if `T` is a (possibly cv-qualified) non-union class type, **its padding bits are initialized to zero bits and** each non-static data member and each base-class subobject is zero-initialized~~ and padding is initialized to zero bits~~;
> - if `T` is a (possibly cv-qualified) union type, **its padding bits are initialized to zero bits and** the object's first non-static named data member is zero-initialized~~ and padding is initialized to zero bits~~;
> - ...

3. Change 12.2.4 [class.bit] paragraph 1 as follows:

> ...The *constant-expression* shall be an integral constant expression with a value greater than or equal to zero. The value of the integral constant expression may be larger than the number of bits in the object representation (6.9 [basic.types]) of the bit-field's type; in such cases the extra bits are ~~used as~~ padding bits **(6.9 [basic.types])**~~and do not participate in the value representation (6.9 [basic.types]) of the bit-field~~. Allocation of bit-fields...

4. Change 6.9.1 [basic.fundamental] paragraph 1 as follows:

> ...For narrow character types, all bits of the object representation participate in the value representation. [*Note:* A bit-field of narrow character type whose length is larger than the number of bits in the object representation of that type has padding bits; see ~~12.2.4 [class.bit]~~ **6.9 [basic.types]**. —*end note*] For unsigned narrow character types...

2273. Inheriting constructors vs implicit default constructor

Section: 15.1 [class.ctor] **Status:** DRWP **Submitter:** Richard Smith **Date:** 2016-06-17

[Voted into the WP at the July, 2017 meeting.]

In an example like

```
struct A { A(int = 0); };
struct B : A { using A::A; };
B b0(0); // #1
B b;     // #2
```

Is #2 valid (presumably calling the constructor inherited from `A`, or ill-formed due to ambiguity with 's implicit default constructor?

Proposed resolution (May, 2017):

1. Change 10.3.3 [namespace.udecl] paragraph 16 as follows:

For the purpose of **forming a set of candidates during** overload resolution, the functions that are introduced by a *using-declaration* into a derived class are treated as though they were members of the derived class. In particular, the implicit `this` parameter shall be treated as if it were a pointer to the derived class rather than to the base class. This has no effect on the type of the function, and in all other respects the function remains a member of the base class. Likewise, constructors that are introduced by a *using-declaration* are treated as though they were constructors of the derived class when looking up the constructors of the derived class (6.4.3.1 [class.qual]) or forming a set of overload candidates (16.3.1.3 [over.match.ctor], 16.3.1.4 [over.match.copy], 16.3.1.7 [over.match.list]). If such a constructor is selected to perform the initialization of an object of class type, all subobjects other than the base class from which the constructor originated are implicitly initialized (15.6.3 [class.inhctor.init]). **[*Note:* A member of a derived class is sometimes preferred to a member of a base class if they would otherwise be ambiguous (16.3.3 [over.match.best]). —*end note*]**

2. Insert the following as a new bullet following 16.3.3 [over.match.best] bullet 1.7:

   - ...
   - `F1` and `F2` are function template specializations, and the function template for `F1` is more specialized than the template for `F2` according to the partial ordering rules described in 17.6.6.2 [temp.func.order], or, if not that,
   - `F1` **is a constructor for a class** `D`**,** `F2` **is a constructor for a base class** `B` **of** `D`**, and for all arguments the corresponding parameters of** `F1` **and** `F2` **have the same type. [*Example:***

     ```
     struct A {
       A(int = 0);
     };

     struct B: A {
       using A::A;
       B();
     };

     int main() {
       B b;  // OK, B::B()
     }
     ```

     ***—end example*], or, if not that,**

   - `F1` is generated from a *deduction-guide* (16.3.1.8 [over.match.class.deduct])...

This resolution also resolves issue 2277.

1299. "Temporary objects" vs "temporary expressions"

Section: 15.2 [class.temporary] **Status:** DRWP **Submitter:** Johannes Schaub **Date:** 2011-04-16

[Voted into the WP at the July, 2017 meeting as document P0727R0.]

The Standard is insufficiently precise in dealing with temporaries. It is not always clear when the term "temporary" is referring to an expression whose result is a prvalue and when it is referring to a temporary object.

(See also issue 1568.)

Proposed resolution (February, 2014) [SUPERSEDED]:

The resolution is contained in document N3918.

This resolution also resolves issues 1651 and 1893.

Additional note, November, 2014:

Concerns have been raised that the meaning of "corresponding temporary object" is not clear enough in the proposed wording. In addition, N3918 says that it resolves issue 1300, but that issue is 1) marked as a duplicate of issue 914 and 2) the subject of continuing deliberations in EWG. This issue is being returned to "review" status to allow CWG to address these concerns.

Proposed resolution (March, 2017):

1. Change 8 [expr] paragraph 12 as follows:

   ...is applied. [*Note:* If the expression is an lvalue of class type, it must have a volatile copy constructor to initialize the temporary **object** that is the result object of the lvalue-to-rvalue conversion. —*end note*] The glvalue expression...

2. Change 15.2 [class.temporary] paragraph 6 as follows:

   The third context is when a reference is bound to a temporary **object**.[116] The temporary **object** to which the reference is bound or the temporary **object** that is the complete object of a subobject to which the reference is bound persists for the lifetime of the reference **if the glvalue referring to the temporary object was obtained through one of the following:**

   - **a temporary materialization conversion (7.4 [conv.rval]),**
   - **(** ***expression*** **), where** ***expression*** **is one of these expressions,**

- **subscripting (8.2.1 [expr.sub]) of an array operand, where that operand is one of these expressions,**
- **a class member access (8.2.5 [expr.ref]) using the `.` operator where the left operand is one of these expressions and the right operand designates a non-static data member of non-reference type,**
- **a pointer-to-member operation (8.5 [expr.mptr.oper]) using the `.*` operator where the left operand is one of these expressions and the right operand is a pointer to data member of non-reference type,**
- **a `const_cast` (8.2.11 [expr.const.cast]), `static_cast` (8.2.9 [expr.static.cast]), `dynamic_cast` (8.2.7 [expr.dynamic.cast]), or `reinterpret_cast` (8.2.10 [expr.reinterpret.cast]) converting a glvalue operand that is one of these expressions to a glvalue referring to that operand,**
- **a conditional expression (8.16 [expr.cond]) that is a glvalue where the second or third operand is one of these expressions, or**
- **a comma expression (8.19 [expr.comma]) that is a glvalue where the right operand is one of these expressions.**

**[*Example:*

```
template<typename T> using id = T;

int&& a = id<int[3]>{1, 2, 3}[i];          // temporary array has same lifetime as a
const int& b = static_cast<const int&>(0); // temporary int has same lifetime as b
int&& c = cond ? id<int[3]>{1, 2, 3}[i] : static_cast<int&&>(0);
                                           // exactly one of the two temporaries is lifetime-extended
```

—end example] [*Note:* **If a temporary object has a reference member initialized by another temporary object, lifetime extension applies recursively to such a member's initializer. [*Example:***

```
struct S {
  const int& m;
};
const S& s = S{1};  // both S and int temporaries have lifetime of s
```

—end example] *—end note*]**

~~except~~ **The exceptions to this lifetime rule are**:

- A temporary object bound to a reference parameter...

3. Change 16.3.1.4 [over.match.copy] bullet 1.2 as follows:
   - When the type of the initializer expression is a class type "*cv* `S`", the non-explicit conversion functions of `S` and its base classes are considered. When initializing a temporary **object (12.2 [class.mem])** to be bound to the first parameter of a constructor...
4. Change 20.5.4.9 [res.on.arguments] bullet 1.3 as follows:
   - ...[*Note:* If a program casts an lvalue to an xvalue while passing that lvalue to a library function (e.g. by calling the function with the argument std::move(x)), the program is effectively asking that function to treat that lvalue as a temporary **object**. The implementation is free...
5. Change 23.11.2.3.3 [util.smartptr.weak.assign] paragraph 2 as follows:

   Remarks: The implementation may meet the effects (and the implied guarantees) via different means, without creating a temporary **object**.
6. Change the footnote in 29.7.2.1 [template.valarray.overview] paragraph 1 as follows:

   ...generalized subscript operators. [*Footnote:* The intent is to specify an array template that hass the minimum functionality necessary to address aliasing ambiguities and the proliferation of ~~temporaries~~ **temporary objects**. Thus... — *end footnote*]
7. Change the last bullet of C.2.16 [diff.cpp03.input.output] as follows:
   - initializing a `const bool &` which would bind to a temporary **object**.

This resolution also resolves issues 943 and 1076.

2290. Unclear specification for overload resolution and deleted special member functions

Section: 16.3.1 [over.match.funcs] **Status:** DRWP **Submitter:** Howard Hinnant **Date:** 2016-07-21

[Voted into the WP at the July, 2017 meeting.]

According to 16.3.1 [over.match.funcs] paragraph 8,

> A defaulted move constructor or assignment operator (15.8 [class.copy]) that is defined as deleted is excluded from the set of candidate functions in all contexts.

It is unclear whether this is intended to apply to all defaulted assignment operators or only move assignment operators.

Proposed resolution (April, 2017):

Change 16.3.1 [over.match.funcs] paragraph 8 as follows:

> A defaulted move ~~constructor or assignment operator~~ **special function** (15.8 [class.copy]) that is defined as deleted is excluded from the set of candidate functions in all contexts.

2277. Ambiguity inheriting constructors with default arguments

Section: 16.3.3.2 [over.ics.rank] **Status:** DRWP **Submitter:** Richard Smith **Date:** 2016-06-23

[Voted into the WP at the July, 2017 meeting.]

In an example like:

```
struct A {
  A(int, int = 0);
  void f(int, int = 0);
};
struct B : A {
  B(int); using A::A;
  void f(int); using A::f;
}
```

calls to `B(int)` and `B::f(int)` are ambiguous, because they could equally call the version inherited from the base class. This doesn't match the intent in 10.3.3 [namespace.udecl], which usually makes derived-class functions take precedence over ones from a base class.

The above patterns are not common, although they sometimes cause breakage when refactoring a base class. However, P0136R1 brings this into sharp focus, because it causes the rejection of the following formerly-valid and very reasonable code:

```
struct A {
  A(int = 0);
};
struct B : A {
  using B::B;
};
B b;
```

Proposed resolution (May, 2017):

This issue is resolved by the resolution of issue 2273.

1704. Type checking in explicit instantiation of variable templates

Section: 17.8.2 [temp.explicit] **Status:** DRWP **Submitter:** Richard Smith **Date:** 2013-06-20

[Voted into the WP at the July, 2017 meeting.]

It is not clear whether the following is well-formed or not:

```
template<typename T> int arr[sizeof(T)] = {};
template int arr<int>[];
```

Are we supposed to instantiate the specialization and treat the explicit instantiation declaration as if it were a redeclaration (in which case the omitted array bound would presumably be OK), or is the type of the explicit instantiation declaration required to exactly match the type that the instantiated specialization has (in which case the omitted bound would presumably not be OK)? Or something else?

(See also issue 1728.)

Proposed resolution (May, 2017):

1. Change 17.8.2 [temp.explicit] paragraph 3 as follows:

> If the explicit instantiation is for a class or member class, the *elaborated-type-specifier* in the *declaration* shall include a *simple-template-id*; **otherwise, the *declaration* shall be a *simple-declaration* whose *init-declarator-list* comprises a single *init-declarator* that does not have an *initializer***. If the explicit instantiation is for a function or member function, the *unqualified-id* in the ~~*declaration*~~ ***declarator*** shall be either a *template-id* or, where all template arguments can be deduced, a *template-name* or *operator-function-id*. [*Note:* The declaration may declare a *qualified-id*, in which case the *unqualified-id* of the *qualified-id* must be a *template-id*. *—end note*] If the explicit instantiation is for a member function, a member class or a static data member of a class template specialization, the name of the class template specialization in the *qualified-id* for the member name shall be a *simple-template-id*. If the explicit instantiation is for a variable **template specialization**, the *unqualified-id* in the ~~declaration~~ **declarator** shall be a ***simple-****template-id*. An explicit instantiation shall appear in an enclosing namespace of its template. If the name declared in the explicit instantiation is an unqualified name, the explicit instantiation shall appear in the namespace where its template is declared or, if that namespace is inline (10.3.1 [namespace.def]), any namespace from its enclosing namespace set. [*Note:*...

2. Add the following as a new paragraph following 17.8.2 [temp.explicit] paragraph 4:

The *declaration* in an *explicit-instantiation* and the *declaration* produced by the corresponding substitution into the templated function, variable, or class are two declarations of the same entity. [*Note:* These declarations are required to have matching types as specified in 6.5 [basic.link], except as specified in 18.4 [except.spec]. [*Example:*

```
template<typename T> T var = {};
template float var<float>;    // OK, instantiated variable has type float
template int var<int[16]>[]; // OK, absence of major array bound is permitted
template int *var<int>;       // error: instantiated variable has type int

template<typename T> auto av = T();
template int av<int>;         // OK, variable with type int can be redeclared with type auto

template<typename T> auto f() {}
template void f<int>();       // error: function with deduced return type redeclared with non-deduced return type (10.1.7.4 [dcl.spec.auto])
```

***—end example*] *—end note*] Despite its syntactic form, the *declaration* in an *explicit-instantiation* for a variable is not itself a definition and does not conflict with the definition instantiated by an explicit instantiation definition for that variable.**

3. Change 17.8.2 [temp.explicit] paragraph 10 as follows:

> Except for inline functions and variables, declarations with types deduced from their initializer or return value (10.1.7.4 [dcl.spec.auto]), `const` variables of literal types, variables of reference types, and class template specializations, explicit instantiation declarations have the effect of suppressing the implicit instantiation of the **definition of the** entity to which they refer. [*Note:*...

This resolution also resolves issue 1728.

1728. Type of an explicit instantiation of a variable template

Section: 17.8.2 [temp.explicit] **Status:** DRWP **Submitter:** Larisse Voufo **Date:** 2013-08-05

[Voted into the WP at the July, 2017 meeting.]

It is not clear to what extent the type in an explicit instantiation must match that of a variable template. For example:

```
template<typename T> T var = T();
template float var<float>;   // #1.
template int* var<int>;      // #2.
template auto var<char>;     // #3.
```

(See also issue 1704.)

Proposed resolution (May, 2017):

This issue is resolved by the resolution of issue 1704.

Issues with "WP" Status

Issues with "CD1" Status

---

663. Valid Cyrillic identifier characters

Section: _N2691_.E [extendid] **Status:** CD1 **Submitter:** Steve Clamage **Date:** 30 November 2007

[Voted into the WP at the June, 2008 meeting.]

The C99 and C++ Standards disagree about the validity of two Cyrillic characters for use in identifiers. C++ (_N2691_.E [extendid]) says that 040d is valid in an identifier but that 040e is not; C99 (Annex D) says exactly the opposite. In fact, both characters should be accepted in identifiers; see the Unicode chart.

Proposed resolution (February, 2008):

The reference in paragraph 2 should be changed to ISO/IEC TR 10176:2003 and the table should be changed to conform to the one in that document (beginning on page 34).

---

122. *template-id*s as *unqualified-id*s

Section: _N4567_.5.1.1 [expr.prim.general] **Status:** CD1 **Submitter:** Mike Miller **Date:** 3 June 1999

[Moved to DR at 10/01 meeting.]

_N4567_.5.1.1 [expr.prim.general] paragraph 11 reads,

> A *template-id* shall be used as an *unqualified-id* only as specified in 17.8.2 [temp.explicit] , 17.8 [temp.spec] , and 17.6.5 [temp.class.spec] .

What uses of *template-id*s as *unqualified-id*s is this supposed to prevent? And is the list of referenced sections correct/complete? For instance, what about 17.9.1 [temp.arg.explicit], "Explicit template argument specification?" Does its absence from the list in _N4567_.5.1.1 [expr.prim.general] paragraph 11 mean that "`f<int>()`" is ill-formed?

This is even more confusing when you recall that *unqualified-id*s are contained in *qualified-id*s:

> *qualified-id*: `::`opt *nested-name-specifier* `template`opt *unqualified-id*

Is the wording intending to say "used as an *unqualified-id* that is not part of a *qualified-id*?" Or something else?

Proposed resolution (10/00):

Remove the referenced sentence altogether.

---

125. Ambiguity in `friend` declaration syntax

Section: _N4567_.5.1.1 [expr.prim.general] **Status:** CD1 **Submitter:** Martin von Loewis **Date:** 7 June 1999

[Voted into WP at March 2004 meeting.]

The example below is ambiguous.

```
struct A{
  struct B{};
};

A::B C();

namespace B{
  A C();
}

struct Test {
  friend A::B ::C();
};
```

Here, it is not clear whether the friend declaration denotes `A B::C()` or `A::B C()`, yet the standard does not resolve this ambiguity.

The ambiguity arises since both the *simple-type-specifier* (10.1.7.2 [dcl.type.simple] paragra 1) and an *init-declararator* (11 [dcl.decl] paragraph 1) contain an optional `::` and an optional *nested-name-specifier* (_N4567_.5.1.1 [expr.prim.general] paragraph 1). Therefore, two different ways to analyse this code are possible:

> *simple-type-specifier* = `A::B`
> *init-declarator* = `::C()`
> *simple-declaration* = `friend A::B ::C();`

or

> *simple-type-specifier* = `A`
> *init-declarator* = `::B::C()`
> *simple-declaration* = `friend A ::B::C();`

Since it is a friend declaration, the init-declarator may be qualified, and start with a global scope.

Suggested Resolution: In the definition of *nested-name-specifier*, add a sentence saying that a `::` token immediately following a *nested-name-specifier* is always considered as part of the *nested-name-specifier*. Under this interpretation, the example is ill-formed, and should be corrected as either

```
friend A (::B::C)();   //or
friend A::B (::C)();
```

An alternate suggestion — changing 10.1 [dcl.spec] to say that

> The longest sequence of **tokens** that could possibly be a type name is taken as the *decl-specifier-seq* of a *declaration*.

— is undesirable because it would make the example well-formed rather than requiring the user to disambiguate the declaration explicitly.

Proposed resolution (04/01):

(See below for problem with this, from 10/01 meeting.)

In _N4567_.5.1.1 [expr.prim.general] paragraph 7,

1. Before the grammar for *qualified-id*, start a new paragraph 7a with the text

   A *qualified-id* is an *id-expression* that contains the scope resolution operator `::`.

2. Following the grammar fragment, insert the following:

   The longest sequence of tokens that could form a *qualified-id* constitutes a single *qualified-id*. [*Example:*

   ```
   // classes C, D; functions F, G, namespace N; non-class type T
   friend C ::D::F();   // ill-formed, means friend (C::D::F)();
   friend C (::D::F)(); // well-formed
   friend N::T ::G();   // ill-formed, means friend (N::T::G)();
   friend N::T (::G)(); // well-formed
   ```

   —*end example*]

3. Start a new paragraph 7b following the example.

(This resolution depends on that of issue 215.)

Notes from 10/01 meeting:

It was pointed out that the proposed resolution does not deal with cases like `X::Y` where `X` is a type but not a class type. The working group reaffirmed its decision that the disambiguation should be syntactic only, i.e., it should depend only on whether or not the name is a type.

Jason Merrill :

At the Seattle meeting, I suggested that a solution might be to change the *class-or-namespace-name* in the *nested-name-specifier* rule to just be "identifier"; there was some resistance to this idea. FWIW, I've tried this in g++. I had to revise the idea so that only the second and subsequent names were open to being any identifier, but that seems to work just fine.

So, instead of

nested-name-specifier:
class-or-namespace-name `::` *nested-name-specifier*$_{\text{opt}}$
class-or-namespace-name `::` *template nested-name-specifier*

it would be

nested-name-specifier:
type-or-namespace-name `::` (per issue 215)
nested-name-specifier identifier `::`
nested-name-specifier `template` *template-id* `::`

Or some equivalent but right-associative formulation, if people feel that's important, but it seems irrelevant to me.

Clark Nelson :

Personally, I prefer the left-associative rule. I think it makes it easier to understand. I was thinking about this production a lot at the meeting, considering also some issues related to 301. My formulation was getting kind of ugly, but with a left-associative rule, it gets a lot nicer.

Your proposal isn't complete, however, as it doesn't allow template arguments without an explicit template keyword. You probably want to add an alternative for:

nested-name-specifier type-or-namespace-name `::`

There is admittedly overlap between this alternative and

nested-name-specifier identifier `::`

but I think they're both necessary.

Notes from the 4/02 meeting:

The changes look good. Clark Nelson will merge the two proposals to produce a single proposed resolution.

Proposed resolution (April 2003):

nested-name-specifier is currently defined in _N4567_.5.1.1 [expr.prim.general] paragraph 7 as:

nested-name-specifier:
class-or-namespace-name `::` *nested-name-specifier*$_{opt}$
class-or-namespace-name `::` `template` *nested-name-specifier*
class-or-namespace-name:
class-name
namespace-name

The proposed definition is instead:

nested-name-specifier:
type-name `::`
namespace-name `::`
nested-name-specifier identifier `::`
nested-name-specifier `template`$_{opt}$ *template-id* `::`

Issue 215 is addressed by using *type-name* instead of *class-name* in the first alternative. Issue 125 (this issue) is addressed by using *identifier* instead of anything more specific in the third alternative. Using left association instead of right association helps eliminate the need for *class-or-namespace-name* (or *type-or-namespace-name*, as suggested for issue 215).

It should be noted that this formulation also rules out the possibility of `A::template B::`, i.e. using the `template` keyword without any template arguments. I think this is according to the purpose of the `template` keyword, and that the former rule allowed such a construct only because of the difficulty of formulation of a right-associative rule that would disallow it. But I wanted to be sure to point out this implication.

Notes from April 2003 meeting:

See also issue 96.

The proposed change resolves only part of issue 215.

---

357. Definition of signature should include name

Section: 3 [intro.defs] **Status:** CD1 **Submitter:** Steve Clamage **Date:** 26 May 2002

[Voted into WP at April, 2007 meeting.]

Section 3 [intro.defs], definition of "signature" omits the function name as part of the signature. Since the name participates in overload resolution, shouldn't it be included in the definition? I didn't find a definition of signature in the ARM, but I might have missed it.

Fergus Henderson: I think so. In particular, _N4140_.17.6.4.3.2 [global.names] reserves certain "function signatures" for use by the implementation, which would be wrong unless the signature includes the name.

> -2- Each global function signature declared with external linkage in a header is reserved to the implementation to designate that function signature with external linkage.
>
> -5- Each function signature from the Standard C library declared with external linkage is reserved to the implementation for use as a function signature with both extern "C" and extern "C++" linkage, or as a name of namespace scope in the global namespace.

Other uses of the term "function signature" in the description of the standard library also seem to assume that it includes the name.

James Widman:

Names don't participate in overload resolution; name lookup is separate from overload resolution. However, the word "signature" is not used in clause 16 [over]. It *is* used in linkage and declaration matching (e.g., 17.6.6.1 [temp.over.link]). This suggests that the name and scope of the function should be part of its signature.

Proposed resolution (October, 2006):

1. Replace 3 [intro.defs] "**signature**" with the following:

   > the name and the parameter-type-list (11.3.5 [dcl.fct]) of a function, as well as the class or namespace of which it is a member. If a function or function template is a class member its signature additionally includes the *cv*-qualifiers (if any) on the function or function template itself. The signature of a function template additionally includes its return type and its template parameter list. The signature of a function template specialization includes the signature of the template of which it is a specialization and its template arguments (whether explicitly specified or deduced). [*Note:* Signatures are used as a basis for name-mangling and linking. —*end note*]

2. Delete paragraph 3 and replace the first sentence of 17.6.6.1 [temp.over.link] as follows:

   > ~~The signature of a function template specialization consists of the signature of the function template and of the actual template arguments (whether explicitly specified or deduced).~~
   >
   > The signature of a function template ~~consists of its function signature, its return type and its template parameter list~~ **is defined in 3 [intro.defs]**. The names of the template parameters are significant...

(See also issue 537.)

---

537. Definition of "signature"

Section: 3 [intro.defs] **Status:** CD1 **Submitter:** Daveed Vandevoorde **Date:** 12 October 2005

[Voted into WP at April, 2007 meeting.]

The standard defines "signature" in two places: 3 [intro.defs] and 17.6.6.1 [temp.over.link] paragraphs 3-4. The former seems to be meant as a formal definition (I think it's the only place covering the nontemplate case), yet it lacks some bits mentioned in the latter (specifically, the notion of a "signature of a function template," which is part of every signature of the associated function template specializations).

Also, I think the 3 [intro.defs] words "the information about a function that participates in overload resolution" isn't quite right either. Perhaps, "the information about a function that distinguishes it in a set of overloaded functions?"

Eric Gufford:

In 3 [intro.defs] the definition states that "Function signatures do not include return type, because that does not participate in overload resolution," while 17.6.6.1 [temp.over.link] paragraph 4 states "The signature of a function template consists of its function signature, its return type and its template parameter list." This seems inconsistent and potentially confusing. It also seems to imply that two identical function templates with different return types are distinct signatures, which is in direct violation of 16.3 [over.match]. 17.6.6.1 [temp.over.link] paragraph 4 should be amended to include verbiage relating to overload resolution.

Either return types are included in function signatures, or they're not, across the board. IMHO, they should be included as they are an integral part of the function declaration/definition irrespective of overloads. Then verbiage should be added about overload resolution to distinguish between signatures and overload rules. This would help clarify things, as it is commonly understood that overload resolution is based on function signature.

In short, the term "function signature" should be made consistent, and removed from its (implicit, explicit or otherwise) linkage to overload resolution as it is commonly understood.

James Widman:

The problem is that (a) if you say the return type is part of the signature of a non-template function, then you have overloading but not overload resolution on return types (i.e., what we have now with function templates). I don't think anyone wants to make the language uglier in that way. And (b) if you say that the return type is not part of the signature of a function template, you will break code. Given those alternatives, it's probably best to maintain the status quo (which the implementors appear to have rendered faithfully).

Proposed resolution (September, 2006):

This issue is resolved by the resolution of issue 357.

---

513. Non-class "most-derived" objects

Section: 4.5 [intro.object] **Status:** CD1 **Submitter:** Marc Schoolderman **Date:** 20 Mar 2005

[Voted into WP at April, 2006 meeting.]

The standard uses "most derived object" in some places (for example, 3 [intro.defs] "**dynamic type**," 8.3.5 [expr.delete]) to refer to objects of both class and non-class type. However, 4.5 [intro.object] only formally defines it for objects of class type.

Possible fix: Change the wording in 4.5 [intro.object] paragraph 4 from

> an object of a most derived class type is called a most derived object

to

> an object of a most derived class type, or of non-class type, is called a most derived object

Proposed resolution (October, 2005):

Add the indicated words to 4.5 [intro.object] paragraph 4:

> If a complete object, a data member (12.2 [class.mem]), or an array element is of class type, its type is considered the *most derived* class, to distinguish it from the class type of any base class subobject; an object of a most derived class type**, or of a non-class type,** is called a *most derived object*.

---

637. Sequencing rules and example disagree

Section: 4.6 [intro.execution] **Status:** CD1 **Submitter:** Ofer Porat **Date:** 2 June 2007

[Voted into the WP at the September, 2008 meeting.]

In 4.6 [intro.execution] paragraph 16, the following expression is still listed as an example of undefined behavior:

```
i = ++i + 1;
```

However, it appears that the new sequencing rules make this expression well-defined:

1. The assignment side-effect is required to be sequenced after the value computations of both its LHS and RHS (8.18 [expr.ass] paragraph 1).
2. The LHS (`i`) is an lvalue, so its value computation involves computing the address of `i`.
3. In order to value-compute the RHS (`++i + 1`), it is necessary to first value-compute the lvalue expression `++i` and then do an lvalue-to-rvalue conversion on the result. This guarantees that the incrementation side-effect is sequenced before the computation of the addition operation, which in turn is sequenced before the assignment side effect. In other words, it yields a well-defined order and final value for this expression.

It should be noted that a similar expression

```
i = i++ + 1;
```

is still not well-defined, since the incrementation side-effect remains unsequenced with respect to the assignment side-effect.

It's unclear whether making the expression in the example well-defined was intentional or just a coincidental byproduct of the new sequencing rules. In either case either the example should be fixed, or the rules should be changed.

Clark Nelson: In my opinion, the poster's argument is perfectly correct. The rules adopted reflect the CWG's desired outcome for issue 222. At the Portland meeting, I presented (and still sympathize with) Tom Plum's case that these rules go a little too far in nailing down required behavior; this is a consequence of that.

One way or another, a change needs to be made, and I think we should seriously consider weakening the resolution of issue 222 to keep this example as having undefined behavior. This could be done fairly simply by having the sequencing requirements for an assignment expression depend on whether it appears in an lvalue context.

James Widman: How's this for a possible re-wording?

> In all cases, the side effect of the assignment expression is sequenced after the value computations of the right and left operands. Furthermore, if the assignment expression appears in a context where an lvalue is required, the side effect of the assignment expression is sequenced before its value computation.

Notes from the February, 2008 meeting:

There was no real support in the CWG for weakening the resolution of issue 222 and returning the example to having undefined behavior. No one knew of an implementation that doesn't already do the (newly) right thing for such an example, so there was little motivation to go out of our way to increase the domain of undefined behavior. So the proposed resolution is to change the example to one that definitely does have undependable behavior in existing practice, and undefined behavior under the new rules.

Also, the new formulation of the sequencing rules approved in Oxford contained the wording that by and large resolved issue 222, so with the resolution of this issue, we can also close issue 222.

Proposed resolution (March, 2008):

Change the example in 4.6 [intro.execution] paragraph 16 as follows:

```
i = v[i++];             // the behavior is undefined
i = 7, i++, i++;        // i becomes 9
i = ~~++i~~ i++ + 1;    // the behavior is undefined
i = i + 1;              // the value of i is incremented
```

This resolution also resolves issue 222.

---

639. What makes side effects "different" from one another?

Section: 4.6 [intro.execution] **Status:** CD1 **Submitter:** James Widman **Date:** 26 July 2007

[Voted into the WP at the September, 2008 meeting.]

Is the behavior undefined in the following example?

```
void f() {
     int n = 0;
     n = --n;
}
```

4.6 [intro.execution] paragraph 16 says,

> If a side effect on a scalar object is unsequenced relative to either a different side effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined.

It's not clear to me whether the two side-effects in `n=--n` are "different." As far as I can tell, it seems that both side-effects involve the assignment of -1 to n, which in a sense makes them non-"different." But I don't know if that's the intent. Would it be better to say "another" instead of "a different?"

On a related note, can we include this example to illustrate?

```
void f( int, int );
void g( int a ) { f( a = -1, a = -1 ); } // Undefined?
```

Proposed resolution (March, 2008):

Change 4.6 [intro.execution] paragraph 16 as follows:

> ...If a side effect on a scalar object is unsequenced relative to either ~~a different~~ **another** side effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined. [*Example:*
>
> ```
> void f(int, int);
> void g(int i, int* v) {
>     i = v[i++];         // the behavior is undefined
>     i = 7, i++, i++;    // i becomes 9
>
>     i = ++i + 1;        // the behavior is undefined
>     i = i + 1;          // the value of i is incremented
>
>      f(i = -1, i = -1);   // the behavior is undefined
> }
> ```
>
> —*end example*] When calling...

---

362. Order of initialization in instantiation units

Section: 5.2 [lex.phases] **Status:** CD1 **Submitter:** Mark Mitchell **Date:** 2 July 2002

[Voted into WP at March 2004 meeting.]

Should this program do what its author obviously expects? As far as I can tell, the standard says that the point of instantiation for Fib<n-1>::Value is the same as the point of instantiation as the enclosing specialization, i.e., Fib<n>::Value. What in the standard actually says that these things get initialized in the right order?

```
template<int n>
struct Fib { static int Value; };

template <>
int Fib<0>::Value = 0;

template <>
int Fib<1>::Value = 1;

template<int n>
int Fib<n>::Value = Fib<n-1>::Value + Fib<n-2>::Value;

int f ()
{
  return Fib<40>::Value;
}
```

John Spicer: My opinion is that the standard does not specify the behavior of this program. I thought there was a core issue related to this, but I could not find it. The issue that I recall proposed tightening up the static initialization rules to make more cases well defined.

Your comment about point of instantiation is correct, but I don't think that really matters. What matters is the order of execution of the initialization code at execution time. Instantiations don't really live in "translation units" according to the standard. They live in "instantiation units", and the handling of instantiation units in initialization is unspecified (which should probably be another core issue). See 5.2 [lex.phases] paragraph 8.

Notes from October 2002 meeting:

We discussed this and agreed that we really do mean the the order is unspecified. John Spicer will propose wording on handling of instantiation units in initialization.

Proposed resolution (April 2003):

TC1 contains the following text in 6.6.2 [basic.start.static] paragraph 1:

> Objects with static storage duration defined in namespace scope in the same translation unit and dynamically initialized shall be initialized in the order in which their definition appears in the translation unit.

This was revised by issue 270 to read:

> Dynamic initialization of an object is either ordered or unordered. Explicit specializations and definitions of class template static data members have ordered initialization. Other class template static data member instances have unordered initialization. Other objects defined in namespace scope have ordered initialization. Objects defined within a single translation unit and with ordered initialization shall be initialized in the order of their definitions in the translation unit. The order of initialization is unspecified for objects with unordered initialization and for objects defined in different translation units.

This addresses this issue but while reviewing this issue some additional changes were suggested for the above wording:

> Dynamic initialization of an object is either ordered or unordered. **Definitions of explicitly specialized** ~~Explicit specializations and definitions of~~ class template static data members have ordered initialization. Other class template static data member**s** **(i.e., implicitly or explicitly instantiated specializations)** ~~instances~~ have unordered initialization. Other objects defined in

namespace scope have ordered initialization. Objects defined within a single translation unit and with ordered initialization shall be initialized in the order of their definitions in the translation unit. The order of initialization is unspecified for objects with unordered initialization and for objects defined in different translation units.

---

558. Excluded characters in universal character names

Section: 5.3 [lex.charset] **Status:** CD1 **Submitter:** Daveed Vandevoorde **Date:** 8 February 2006

[Moved to DR at October 2007 meeting.]

C99 and C++ differ in their approach to universal character names (UCNs).

Issue 248 already covers the differences in UCNs allowed for identifiers, but a more fundamental issue is that of UCNs that correspond to codes reserved by ISO 10676 for surrogate pair forms.

Specifically, C99 does not allow UCNs whose short names are in the range 0xD800 to 0xDFFF. I think C++ should have the same constraint. If someone really wants to place such a code in a character or string literal, they should use a hexadecimal escape sequence instead, for example:

```
wchar_t  w1 = L'\xD900'; // Okay.
wchar_t  w2 = L'\uD900'; // Error, not a valid character.
```

(Compare 6.4.3 paragraph 2 in ISO/IEC 9899/1999 with 5.3 [lex.charset] paragraph 2 in the C++ standard.)

Proposed resolution (October, 2007):

This issue is resolved by the adoption of paper J16/07-0030 = WG21 N2170.

---

505. Conditionally-supported behavior for unknown character escapes

Section: 5.13.3 [lex.ccon] **Status:** CD1 **Submitter:** Mike Miller **Date:** 14 Apr 2005

[Voted into WP at the October, 2006 meeting.]

The current wording of 5.13.3 [lex.ccon] paragraph 3 states,

> If the character following a backslash is not one of those specified, the behavior is undefined.

Paper J16/04-0167=WG21 N1727 suggests that such character escapes be ill-formed. In discussions at the Lillehammer meeting, however, the CWG felt that the newly-approved category of conditionally-supported behavior would be more appropriate.

Proposed resolution (April, 2006):

Change the next-to-last sentence of 5.13.3 [lex.ccon] paragraph 3 from:

> If the character following a backslash is not one of those specified, the behavior is undefined.

to:

> Escape sequences in which the character following the backslash is not listed in Table 6 are conditionally-supported, with implementation-defined semantics.

---

309. Linkage of entities whose names are not simply identifiers, in introduction

Section: 6 [basic] **Status:** CD1 **Submitter:** Mike Miller **Date:** 17 Sep 2001

[Voted into the WP at the June, 2008 meeting.]

6 [basic] paragraph 8, while not incorrect, does not allow for linkage of operators and conversion functions. It says:

> An identifier used in more than one translation unit can potentially refer to the same entity in these translation units depending on the linkage (6.5 [basic.link]) of the identifier specified in each translation unit.

Proposed Resolution (November, 2006):

This issue is resolved by the proposed resolution of issue 485.

---

485. What is a "name" ?

Section: 6 [basic] **Status:** CD1 **Submitter:** Gabriel Dos Reis **Date:** 9 Nov 2004

[Voted into the WP at the June, 2008 meeting.]

Clause 6 [basic] paragraph 4 says:

> A *name* is a use of an identifier (5.10 [lex.name]) that denotes an entity or label (9.6.4 [stmt.goto], 9.1 [stmt.label]).

Just three paragraphs later, it says

> Two names are the *same* if
>
> - they are identifiers composed of the same character sequence; or
> - they are the names of overloaded operator functions formed with the same operator; or
> - they are the names of user-defined conversion functions formed with the same type.

The last two bullets contradict the definition of *name* in paragraph 4 because they are not identifiers.

This definition affects other parts of the Standard, as well. For example, in 6.4.2 [basic.lookup.argdep] paragraph 1,

> When an unqualified name is used as the *postfix-expression* in a function call (8.2.2 [expr.call]), other namespaces not considered during the usual unqualified lookup (6.4.1 [basic.lookup.unqual]) may be searched, and in those namespaces, namespace-scope friend function declarations (14.3 [class.friend]) not otherwise visible may be found.

With the current definition of *name*, argument-dependent lookup apparently does not apply to function-notation calls to overloaded operators.

Another related question is whether a *template-id* is a name or not and thus would trigger an argument-dependent lookup. Personally, I have always viewed a *template-id* as a name, just like `operator+`.

Proposed Resolution (November, 2006):

1. Change clause 6 [basic] paragraphs 3-8 as follows:

> 3. An *entity* is a value, object, ~~subobject, base class subobject, array element,~~ variable, **reference,** function, ~~instance of a function,~~ enumerator, type, class member, template, **template specialization,** namespace, or parameter pack.
>
> 4. A *name* is a use of an ~~identifier~~ ***identifier*** (5.10 [lex.name])**,** ***operator-function-id*** **(16.5 [over.oper]),** ***conversion-function-id*** **(15.3.2 [class.conv.fct]), or** ***template-id*** **(17.2 [temp.names])** that denotes an entity or *label* (9.6.4 [stmt.goto], 9.1 [stmt.label]). ~~A *variable* is introduced by the declaration of an object. The variable's name denotes the object.~~
>
> 5. Every name that denotes an entity is introduced by a *declaration*. Every name that denotes a label is introduced either by a `goto` statement (9.6.4 [stmt.goto]) or a *labeled-statement* (9.1 [stmt.label]).
>
> a. **A *variable* is introduced by the declaration of an object. The variable's name denotes the object.**
>
> 6. Some names denote types~~, classes, enumerations,~~ or templates. In general, it is necessary to determine whether or not a name denotes one of these entities before parsing the program that contains it. The process that determines this is called *name lookup* (6.4 [basic.lookup]).
>
> 7. Two names are *the same* if
>
> - they are ~~identifiers~~ ***identifier*****s** composed of the same character sequence; or
> - they are ~~the names of overloaded operator functions~~ ***operator-function-id*****s** formed with the same operator; or
> - they are ~~the names of user-defined conversion functions~~ ***conversion-function-id*****s** formed with the same type~~.~~**, or**
> - **they are** ***template-id*****s that refer to the same class or function (17.5 [temp.type]).**
>
> 8. ~~An identifier~~ **A name** used in more than one translation unit can potentially refer to the same entity in these translation units depending on the linkage (6.5 [basic.link]) of the ~~identifier~~ **name** specified in each translation unit.

2. Change 6.3.7 [basic.scope.class] paragraph 1 item 5 as follows:

> The potential scope of a declaration that extends to or past the end of a class definition also extends to the regions defined by its member definitions, even if the members are defined lexically outside the class (this includes static data member definitions, nested class definitions, member function definitions (including the member function body and any portion of the declarator part of such definitions which follows the ~~identifier~~ ***declarator-id***, including a *parameter-declaration-clause* and any default arguments (11.3.6 [dcl.fct.default]).

[Drafting note: This last change is not really mandated by the issue, but it's another case of "identifier" confusion.]

(This proposed resolution also resolves issue 309.)

261. When is a deallocation function "used?"

Section: 6.2 [basic.def.odr] **Status:** CD1 **Submitter:** Mike Miller **Date:** 7 Nov 2000

[Moved to DR at October 2002 meeting.]

6.2 [basic.def.odr] paragraph 2 says that a deallocation function is "used" by a *new-expression* or *delete-expression* appearing in a potentially-evaluated expression. 6.2 [basic.def.odr] paragraph 3 requires only that "used" functions be defined.

This wording runs afoul of the typical implementation technique for polymorphic *delete-expression*s in which the deallocation function is invoked from the virtual destructor of the most-derived class. The problem is that the destructor must be defined, because it's virtual, and if it contains an implicit reference to the deallocation function, the deallocation function must also be defined, even if there are no relevant *new-expression*s or *delete-expression*s in the program.

For example:

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

struct D: B {
    void operator delete(void*);
    ~D() { }
};
```

Is it required that `D::operator delete(void*)` be defined, even if no `B` or `D` objects are ever created or deleted?

Suggested resolution: Add the words "or if it is found by the lookup at the point of definition of a virtual destructor (15.4 [class.dtor])" to the specification in 6.2 [basic.def.odr] paragraph 2.

Notes from 04/01 meeting:

The consensus was in favor of requiring that any declared non-placement `operator delete` member function be defined if the destructor for the class is defined (whether virtual or not), and similarly for a non-placement `operator new` if a constructor is defined.

Proposed resolution (10/01):

In 6.2 [basic.def.odr] paragraph 2, add the indicated text:

> An allocation or deallocation function for a class is used by a new expression appearing in a potentially-evaluated expression as specified in 8.3.4 [expr.new] and 15.5 [class.free]. A deallocation function for a class is used by a delete expression appearing in a potentially-evaluated expression as specified in 8.3.5 [expr.delete] and 15.5 [class.free]. **A non-placement allocation or deallocation function for a class is used by the definition of a constructor of that class. A non-placement deallocation function for a class is used by the definition of the destructor of that class, or by being selected by the lookup at the point of definition of a virtual destructor (15.4 [class.dtor]). [Footnote: An implementation is not required to call allocation and deallocation functions from constructors or destructors; however, this is a permissible implementation technique.]**

---

289. Incomplete list of contexts requiring a complete type

Section: 6.2 [basic.def.odr] **Status:** CD1 **Submitter:** Mike Miller **Date:** 25 May 2001

[Moved to DR at October 2002 meeting.]

6.2 [basic.def.odr] paragraph 4 has a note listing the contexts that require a class type to be complete. It does not list use as a base class as being one of those contexts.

Proposed resolution (10/01):

In 6.2 [basic.def.odr] paragraph 4 add a new bullet at the end of the note as the next-to-last bullet:

- a class with a base class of type `T` is defined (13 [class.derived]), or

---

433. Do elaborated type specifiers in templates inject into enclosing namespace scope?

Section: 6.3.2 [basic.scope.pdecl] **Status:** CD1 **Submitter:** Daveed Vandevoorde **Date:** 2 September 2003

[Voted into WP at March 2004 meeting.]

Consider the following translation unit:

```
template<class T> struct S {
  void f(union U*);  // (1)
};
template<class T> void S<T>::f(union U*) {}  // (2)
U *p;  // (3)
```

Does (1) introduce U as a visible name in the surrounding namespace scope?

If not, then (2) could presumably be an error since the "union U" in that definition does not find the same type as the declaration (1).

If yes, then (3) is OK too. However, we have gone through much trouble to allow template implementations that do not pre-parse the template definitions, but requiring (1) to be visible would change that.

A slightly different case is the following:

```
template<typename> void f() { union U *p; }
U *q;  // Should this be valid?
```

Notes from October 2003 meeting:

There was consensus that example 1 should be allowed. (Compilers already parse declarations in templates; even MSVC++ 6.0 accepts this case.) The vote was 7-2.

Example 2, on the other hand, is wrong; the union name goes into a block scope anyway.

Proposed resolution:

In 6.3.2 [basic.scope.pdecl] change the second bullet of paragraph 5 as follows:

> for an *elaborated-type-specifier* of the form
>
> ```
> class-key identifier
> ```
>
> if the *elaborated-type-specifier* is used in the *decl-specifier-seq* or *parameter-declaration-clause* of a function defined in namespace scope, the identifier is declared as a *class-name* in the namespace that contains the declaration; otherwise, except as a friend declaration, the identifier is declared in the smallest non-class, non-function-prototype scope that contains the declaration. **[Note: These rules also apply within templates.]** [Note: ...]

---

432. Is injected class name visible in base class specifier list?

Section: 6.3.7 [basic.scope.class] **Status:** CD1 **Submitter:** Daveed Vandevoorde **Date:** 29 August 2003

[Voted into WP at March 2004 meeting.]

Consider the following example (inspired by a question from comp.lang.c++.moderated):

```
template<typename> struct B {};
template<typename T> struct D: B<D> {};
```

Most (all?) compilers reject this code because D is handled as a template name rather than as the injected class name.

12 [class]/2 says that the injected class name is "inserted into the scope of the class."

6.3.7 [basic.scope.class]/1 seems to be the text intended to describe what "scope of a class" means, but it assumes that every name in that scope was introduced using a "declarator". For an implicit declaration such as the injected-class name it is not clear what that means.

So my questions:

1. Should the injected class name be available in the base class specifiers?

   > John Spicer: I do not believe the injected class name should be available in the base specifier. I think the semantics of injected class names should be as if a magic declaration were inserted after the opening "{" of the class definition. The injected class name is a member of the class and members don't exist at the point where the base specifiers are scanned.

2. Do you agree the wording should be clarified whatever the answer to the first question?

   > John Spicer: I believe the 6.3.7 [basic.scope.class] wording should be updated to reflect the fact that not all names come from declarators.

Notes from October 2003 meeting:

We agree with John Spicer's suggested answers above.

Proposed Resolution (October 2003):

The answer to question 1 above is No and no change is required.

For question 1, change 6.3.7 [basic.scope.class] paragraph 1 rule 1 to:

> 1) The potential scope of a name declared in a class consists not only of the declarative region following the name's **point of declaration** ~~declarator~~, but also of all function bodies, default arguments, and constructor *ctor-initializers* in that class (including such things in nested classes). The point of declaration of an *injected-class-name* (clause 12 [class]) is immediately following the opening brace of the class definition.

(Note that this change overlaps a change in issue 417.)

Also change 6.3.2 [basic.scope.pdecl] by adding a new paragraph 8 for the *injected-class-name* case:

The point of declaration for an *injected-class-name* (clause 12 [class]) is immediately following the opening brace of the class definition.

Alternatively this paragraph could be added after paragraph 5 and before the two note paragraphs (i.e. it would become paragraph 5a).

---

139. Error in `friend` lookup example

Section: 6.4.1 [basic.lookup.unqual] **Status:** CD1 **Submitter:** Mike Miller **Date:** 14 Jul 1999

[Moved to DR at 10/01 meeting.]

The example in 6.4.1 [basic.lookup.unqual] paragraph 3 is incorrect:

```
typedef int f;
struct A {
    friend void f(A &);
    operator int();
    void g(A a) {
        f(a);
    }
};
```

Regardless of the resolution of other issues concerning the lookup of names in `friend` declarations, this example is ill-formed (the function and the typedef cannot exist in the same scope).

One possible repair of the example would be to make `f` a class with a constructor taking either `A` or `int` as its parameter.

(See also issues 95, 136, 138, 143, 165, and 166.)

Proposed resolution (04/01):

1. Change the example in 6.4.1 [basic.lookup.unqual] paragraph 3 to read:

```
typedef int f;
namespace N {
    struct A {
        friend int f(A &);
        operator int();
        void g(A a) {
            int i = f(a);
                  // f is the typedef, not the friend function:
                  // equivalent to int(a)
        }
    };
}
```

2. Delete the sentence immediately following the example:

   The expression `f(a)` is a *cast-expression* equivalent to `int(a)`.

---

514. Is the initializer for a namespace member in the scope of the namespace?

Section: 6.4.1 [basic.lookup.unqual] **Status:** CD1 **Submitter:** Mike Miller **Date:** 24 Mar 2005

[Voted into WP at the October, 2006 meeting.]

Is the following code well-formed?

```
namespace N {
  int i;
  extern int j;
}
int N::j = i;
```

The question here is whether the lookup for `i` in the initializer of `N::j` finds the declaration in namespace `N` or not. Implementations differ on this question.

If `N::j` were a static data member of a class, the answer would be clear: both 6.4.1 [basic.lookup.unqual] paragraph 12 and 11.6 [dcl.init] paragraph 11 say that the initializer "is in the scope of the member's class." There is no such provision for namespace members defined outside the namespace, however.

The reasoning given in 6.4.1 [basic.lookup.unqual] may be instructive:

> A name used in the definition of a `static` data member of class `X` (12.2.3.2 [class.static.data]) (after the *qualified-id* of the static member) is looked up as if the name was used in a member function of `X`.

It is certainly the case that a name used in a function that is a member of a namespace is looked up in that namespace (6.4.1 [basic.lookup.unqual] paragraph 6), regardless of whether the definition is inside or outside that namespace. Initializers for namespace members should probably be looked up the same way.

Proposed resolution (April, 2006):

Add a new paragraph following 6.4.1 [basic.lookup.unqual] paragraph 12:

> If a variable member of a namespace is defined outside of the scope of its namespace then any name used in the definition of the variable member (after the *declarator-id*) is looked up as if the definition of the variable member occurred in its namespace. [*Example:*
>
> ```
> namespace N {
>   int i = 4;
>   extern int j;
> }
>
> int i = 2;
>
> int N::j = i;       // N::j == 4
> ```
>
> —*end example*]

---

143. Friends and Koenig lookup

Section: 6.4.2 [basic.lookup.argdep] **Status:** CD1 **Submitter:** Mike Miller **Date:** 21 Jul 1999

[Moved to DR at 4/02 meeting.]

Paragraphs 1 and 2 of 6.4.2 [basic.lookup.argdep] say, in part,

> When an unqualified name is used as the *postfix-expression* in a function call (8.2.2 [expr.call])... namespace-scope friend function declarations (14.3 [class.friend]) not otherwise visible may be found... the set of declarations found by the lookup of the function name [includes] the set of declarations found in the... classes associated with the argument types.

The most straightforward reading of this wording is that if a function of namespace scope (as opposed to a class member function) is declared as a friend in a class, and that class is an associated class in a function call, the friend function will be part of the overload set, even if it is not visible to normal lookup.

Consider the following example:

```
namespace A {
    class S;
};
namespace B {
    void f(A::S);
};
namespace A {
    class S {
        int i;
        friend void B::f(S);
    };
}
void g() {
    A::S s;
    f(s); // should find B::f(A::S)
}
```

This example would seem to satisfy the criteria from 6.4.2 [basic.lookup.argdep] : `A::S` is an associated class of the argument, and `A::S` has a friend declaration of the namespace-scope function `B::f(A::S)`, so Koenig lookup should include `B::f(A::S)` as part of the overload set in the call.

Another interpretation is that, instead of finding the friend declarations in associated classes, one only looks for namespace-scope functions, visible or invisible, in the namespaces of which the the associated classes are members; the only use of the friend declarations in the associated classes is to validate whether an invisible function declaration came from an associated class or not and thus whether it should be included in the overload set or not. By this interpretation, the call `f(s)` in the example will fail, because `B::f(A::S)` is not a member of namespace `A` and thus is not found by the lookup.

Notes from 10/99 meeting: The second interpretation is correct. The wording should be revised to make clear that Koenig lookup works by finding "invisible" declarations in namespace scope and not by finding `friend` declarations in associated classes.

Proposed resolution (04/01): The "associated classes" are handled adequately under this interpretation by 6.4.2 [basic.lookup.argdep] paragraph 3, which describes the lookup in the associated namespaces as including the friend declarations from the associated classes. Other mentions of the associated classes should be removed or qualified to avoid the impression that there is a lookup in those classes:

1. In 6.4.2 [basic.lookup.argdep], change

   > When an unqualified name is used as the *postfix-expression* in a function call (8.2.2 [expr.call]), other namespaces not considered during the usual unqualified lookup (6.4.1 [basic.lookup.unqual]) may be searched, and namespace-scope friend function declarations (14.3 [class.friend]) not otherwise visible may be found.

   to

   > When an unqualified name is used as the *postfix-expression* in a function call (8.2.2 [expr.call]), other namespaces not considered during the usual unqualified lookup (6.4.1 [basic.lookup.unqual]) may be searched, and **in those namespaces,** namespace-scope friend function declarations (14.3 [class.friend]) not otherwise visible may be found.

2. In 6.4.2 [basic.lookup.argdep] paragraph 2, delete the words **and classes** in the following two sentences:

> If the ordinary unqualified lookup of the name finds the declaration of a class member function, the associated namespaces ~~**and classes**~~ are not considered. Otherwise the set of declarations found by the lookup of the function name is the union of the set of declarations found using ordinary unqualified lookup and the set of declarations found in the namespaces ~~**and classes**~~ associated with the argument types.

(See also issues 95, 136, 138, 139, 165, 166, and 218.)

---

218. Specification of Koenig lookup

Section: 6.4.2 [basic.lookup.argdep] **Status:** CD1 **Submitter:** Hyman Rosen **Date:** 28 Mar 2000

[Voted into WP at April, 2007 meeting.]

The original intent of the Committee when Koenig lookup was added to the language was apparently something like the following:

1. The name in the function call expression is looked up like any other unqualified name.
2. If the ordinary unqualified lookup finds nothing or finds the declaration of a (non-member) function, function template, or overload set, argument-dependent lookup is done and any functions found in associated namespaces are added to the result of the ordinary lookup.

This approach is not reflected in the current wording of the Standard. Instead, the following appears to be the status quo:

1. Lookup of an unqualified name used as the *postfix-expression* in the function call syntax always performs Koenig lookup (6.4.1 [basic.lookup.unqual] paragraph 3).
2. Unless ordinary lookup finds a class member function, the result of Koenig lookup always includes the declarations found in associated namespaces (6.4.2 [basic.lookup.argdep] paragraph 2), regardless of whether ordinary lookup finds a declaration and, if so, what kind of entity is found.
3. The declarations from associated namespaces are not limited to functions and template functions by anything in 6.4.2 [basic.lookup.argdep]. However, if Koenig lookup results in more than one declaration and at least one of the declarations is a non-function, the program is ill-formed (10.3.4 [namespace.udir], paragraph 4; although this restriction is in the description of the *using-directive*, the wording applies to any lookup that spans namespaces).

John Spicer: Argument-dependent lookup was created to solve the problem of looking up function names within templates where you don't know which namespace to use because it may depend on the template argument types (and was then expanded to permit use in nontemplates). The original intent only concerned functions. The safest and simplest change is to simply clarify the existing wording to that effect.

Bill Gibbons: I see no reason why non-function declarations should not be found. It would take a special rule to exclude "function objects", as well as pointers to functions, from consideration. There is no such rule in the standard and I see no need for one.

There is also a problem with the wording in 6.4.2 [basic.lookup.argdep] paragraph 2:

> If the ordinary unqualified lookup of the name finds the declaration of a class member function, the associated namespaces and classes are not considered.

This implies that if the ordinary lookup of the name finds the declaration of a data member which is a pointer to function or function object, argument-dependent lookup is still done.

My guess is that this is a mistake based on the incorrect assumption that finding any member other than a member function would be an error. I would just change "class member function" to "class member" in the quoted sentence.

Mike Miller: In light of the issue of "short-circuiting" Koenig lookup when normal lookup finds a non-function, perhaps it should be written as "...finds the declaration of a class member, an object, or a reference, the associated namespaces..."?

Andy Koenig: I think I have to weigh in on the side of extending argument-dependent lookup to include function objects and pointers to functions. I am particularly concerned about [function objects], because I think that programmers should be able to replace functions by function objects without changing the behavior of their programs in fundamental ways.

Bjarne Stroustrup: I don't think we could seriously argue from first principles that [argument-dependent lookup should find only function declarations]. In general, C++ name lookup is designed to be independent of type: First we find the name(s), then, we consider its(their) meaning. 6.4 [basic.lookup] states "The name lookup rules apply uniformly to all names ..." That is an important principle.

Thus, I consider text that speaks of "function call" instead of plain "call" or "application of ()" in the context of koenig lookup an accident of history. I find it hard to understand how 8.2.2 [expr.call] doesn't either disallow all occurrences of `x(y)` where `x` is a class object (that's clearly not intended) or requires koenig lookup for `x` independently of its type (by reference from 6.4 [basic.lookup]). I suspect that a clarification of 8.2.2 [expr.call] to mention function objects is in order. If the left-hand operand of () is a name, it should be looked up using koenig lookup.

John Spicer: This approach causes otherwise well-formed programs to be ill-formed, and it does so by making names visible that might be completely unknown to the author of the program. Using-directives already do this, but argument-dependent lookup is different. You only get names from using-directives if you actually *use* using-directives. You get names from argument-dependent lookup whether you want them or not.

This basically breaks an important reason for having namespaces. You are not supposed to need any knowledge of the names used by a namespace.

But this example breaks if argument-dependent lookup finds non-functions and if the translation unit includes the <list> header somewhere.

```
namespace my_ns {
    struct A {};
    void list(std::ostream&, A&);

    void f() {
        my_ns::A a;
        list(cout, a);
    }
}
```

This really makes namespaces of questionable value if you still need to avoid using the same name as an entity in another namespace to avoid problems like this.

Erwin Unruh: Before we really decide on this topic, we should have more analysis on the impact on programs. I would also like to see a paper on the possibility to overload functions with function surrogates (no, I won't write one). Since such an extension is bound to wait until the next official update, we should not preclude any outcome of the discussion.

I would like to have a change right now, which leaves open several outcomes later. I would like to say that:

> Koenig lookup will find non-functions as well. If it finds a variable, the program is ill-formed. If the primary lookup finds a variable, Koenig lookup is done. If the result contains both functions and variables, the program is ill-formed. [*Note:* A future standard will assign semantics to such a program.]

I myself are not comfortable with this as a long-time result, but it prepares the ground for any of the following long term solutions:

- Do overloading on mixed function/variable sets.
- Ignore variables on Koenig lookup.
- Don't do Koenig lookup if the primary lookup finds a variable.
- Find variables on Koenig lookup and give an error if there is a variable/function mix.

The note is there to prevent compiler vendors to put their own extensions in here.

(See also issues 113 and 143.)

Notes from 04/00 meeting:

Although many agreed that there were valid concerns motivating a desire for Koenig lookup to find non-function declarations, there was also concern that supporting this capability would be more dangerous than helpful in the absence of overload resolution for mixed function and non-function declarations.

A straw poll of the group revealed 8 in favor of Koenig lookup finding functions and function templates only, while 3 supported the broader result.

Notes from the 10/01 meeting:

There was unanimous agreement on one less controversial point: if the normal lookup of the identifier finds a non-function, argument-dependent lookup should not be done.

On the larger issue, the primary point of consensus is that making this change is an extension, and therefore it should wait until the point at which we are considering extensions (which could be very soon). There was also consensus on the fact that the standard as it stands is not clear: some introductory text suggests that argument-dependent lookup finds only functions, but the more detailed text that describes the lookup does not have any such restriction.

It was also noted that some existing implementations (e.g., g++) do find some non-functions in some cases.

The issue at this point is whether we should (1) make a small change to make the standard clear (presumably in the direction of not finding the non-functions in the lookup), and revisit the issue later as an extension, or (2) leave the standard alone for now and make any changes only as part of considering the extension. A straw vote favored option (1) by a strong majority.

Additional Notes (September, 2006):

Recent discussion of this issue has emphasized the following points:

1. The concept of finding function pointers and function objects as part of argument-dependent lookup is not currently under active discussion in the Evolution Working Group.

2. The major area of concern with argument-dependent lookup is finding functions in unintended namespaces. There are current proposals to deal with this concern either by changing the definition of "associated namespace" so that fewer namespaces are considered or to provide a mechanism for enabling or disabling ADL altogether. Although this concern is conceptually distinct from the question of whether ADL finds function pointers and function objects, it is related in the sense that the current rules are perceived as finding too many functions (because of searching too many namespaces), and allowing function pointers and function objects would also increase the number of entities found by ADL.

3. Any expansion of ADL to include function pointers and function objects must necessarily update the overloading rules to specify how they interact with functions and function templates in the overload set. Current implementation experience (g++) is not helpful

in making this decision because, although it performs a uniform lookup and finds non-function entities, it diagnoses an error in overload resolution if non-function entities are in the overload set.

4. There is a possible problem if types are found by ADL: it is not clear that overloading between callable entities (functions, function templates, function pointers, and function objects) and types (where the postfix syntax means a cast or construction of a temporary) is reasonable or useful.

James Widman:

There is a larger debate here about whether ADL should find object names; the proposed wording below is only intended to answer the request for wording to clarify the status quo (option 1 above) and not to suggest the outcome of the larger debate.

Proposed Resolution (October, 2006):

1. Replace the normative text in 6.4.2 [basic.lookup.argdep] paragraph 3 with the following (leaving the text of the note and example unchanged):

   Let X be the lookup set produced by unqualified lookup (6.4.1 [basic.lookup.unqual]) and let Y be the lookup set produced by argument dependent lookup (defined as follows). If X contains

   - a declaration of a class member, or
   - a block-scope function declaration that is not a *using-declaration*, or
   - a declaration that is neither a function nor a function template

   then Y is empty. Otherwise Y is the set of declarations found in the namespaces associated with the argument types as described below. The set of declarations found by the lookup of the name is the union of X and Y.

2. Change 6.4.1 [basic.lookup.unqual] paragraph 4 as indicated:

   When considering an associated namespace, the lookup is the same as the lookup performed when the associated namespace is used as a qualifier (6.4.3.2 [namespace.qual]) except that:

   - Any *using-directive*s in the associated namespace are ignored.
   - Any namespace-scope friend functions **or friend function templates** declared in associated classes are visible within their respective namespaces even if they are not visible during an ordinary lookup (14.3 [class.friend]).
   - **All names except those of (possibly overloaded) functions and function templates are ignored.**

---

403. Reference to a type as a *template-id*

Section: 6.4.2 [basic.lookup.argdep] **Status:** CD1 **Submitter:** John Spicer **Date:** 18 Sep 2003

[Voted into WP at March 2004 meeting.]

Spun off from issue 384.

6.4.2 [basic.lookup.argdep] says:

> If T is a template-id, its associated namespaces and classes are the namespace in which the template is defined; for member templates, the member template's class; the namespaces and classes associated with the types of the template arguments provided for template type parameters (excluding template template parameters); the namespaces in which any template template arguments are defined; and the classes in which any member templates used as template template arguments are defined. [Note: non-type template arguments do not contribute to the set of associated namespaces.]

There is a problem with the term "is a template-id". template-id is a syntactic construct and you can't really talk about a type being a template-id. Presumably, this is intended to mean "If T is the type of a class template specialization ...".

Proposed Resolution (October 2003):

In 6.4.2 [basic.lookup.argdep], paragraph 2, bullet 8, replace

> If T is a *template-id* ...

with

> If T is a class template specialization ...

---

557. Does argument-dependent lookup cause template instantiation?

Section: 6.4.2 [basic.lookup.argdep] **Status:** CD1 **Submitter:** Mike Miller **Date:** 8 February 2006

[Voted into WP at the October, 2006 meeting.]

One might assume from 17.8.1 [temp.inst] paragraph 1 that argument-dependent lookup would require instantiation of any class template specializations used in argument types:

> Unless a class template specialization has been explicitly instantiated (17.8.2 [temp.explicit]) or explicitly specialized (17.8.3 [temp.expl.spec]), the class template specialization is implicitly instantiated when the specialization is referenced in a context that requires a completely-defined object type or when the completeness of the class type affects the semantics of the program.

A complete class type is required to determine the associated classes and namespaces for the argument type (to determine the class's bases) and to determine the friend functions declared by the class, so the completeness of the class type certainly "affects the semantics of the program."

This conclusion is reinforced by the second bullet of 6.4.2 [basic.lookup.argdep] paragraph 2:

- If `T` is a class type (including unions), its associated classes are: the class itself; the class of which it is a member, if any; and its direct and indirect base classes. Its associated namespaces are the namespaces in which its associated classes are defined.

A class template specialization is a class type, so the second bullet would appear to apply, requiring the specialization to be instantiated in order to determine its base classes.

However, bullet 8 of that paragraph deals explicitly with class template specializations:

- If `T` is a class template specialization its associated namespaces and classes are the namespace in which the template is defined; for member templates, the member template's class; the namespaces and classes associated with the types of the template arguments provided for template type parameters (excluding template template parameters); the namespaces in which any template template arguments are defined; and the classes in which any member templates used as template template arguments are defined.

Note that the class template specialization itself is *not* listed as an associated class, unlike other class types, and there is no mention of base classes. If bullet 8 were intended as a supplement to the treatment of class types in bullet 2, one would expect phrasing along the lines of, "In addition to the associated namespaces and classes for all class types..." or some such; instead, bullet 8 reads like a self-contained and complete specification.

If argument-dependent lookup does not cause implicit instantiation, however, examples like the following fail:

```
template <typename T> class C {
    friend void f(C<T>*) { }
};
void g(C<int>* p) {
    f(p);    // found by ADL??
}
```

Implementations differ in whether this example works or not.

Proposed resolution (April, 2006):

1. Change bullet 2 of 6.4.2 [basic.lookup.argdep] paragraph 2 as indicated:

   - If `T` is a class type (including unions), its associated classes are: the class itself; the class of which it is a member, if any; and its direct and indirect base classes. Its associated namespaces are the namespaces ~~in~~ **of** which its associated classes are ~~defined~~ **members**. **Furthermore, if `T` is a class template specialization, its associated namespaces and classes also include: the namespaces and classes associated with the types of the template arguments provided for template type parameters (excluding template template parameters); the namespaces of which any template template arguments are members; and the classes of which any member templates used as template template arguments are members. [*Note:* Non-type template arguments do not contribute to the set of associated namespaces. —*end note*]**

2. Delete bullet 8 of 6.4.2 [basic.lookup.argdep] paragraph 2:

   - ~~If `T` is a class template specialization its associated namespaces and classes are the namespace in which the template is defined; for member templates, the member template's class; the namespaces and classes associated with the types of the template arguments provided for template type parameters (excluding template template parameters); the namespaces in which any template template arguments are defined; and the classes in which any member templates used as template template arguments are defined. [*Note:* non-type template arguments do not contribute to the set of associated namespaces. —*end note*]~~

---

298. `T::x` **when** `T` **is cv-qualified**

Section: 6.4.3.1 [class.qual] **Status:** CD1 **Submitter:** Steve Adamczyk **Date:** 7 Jul 2001

[Voted into WP at April 2003 meeting.]

Can a typedef `T` to a cv-qualified class type be used in a qualified name `T::x`?

```
struct A { static int i; };
typedef const A CA;
int main () {
  CA::i = 0;  // Okay?
}
```

Suggested answer: Yes. All the compilers I tried accept the test case.

Proposed resolution (10/01):

In 6.4.3.1 [class.qual] paragraph 1 add the indicated text:

> If the *nested-name-specifier* of a *qualified-id* nominates a class, the name specified after the *nested-name-specifier* is looked up in the scope of the class (13.2 [class.member.lookup]), except for the cases listed below. The name shall represent one or more members of that class or of one of its base classes (clause 13 [class.derived]). **If the *class-or-namespace-name* of the *nested-name-specifier* names a cv-qualified class type, it nominates the underlying class (the cv-qualifiers are ignored).**

Notes from 4/02 meeting:

There is a problem in that *class-or-namespace-name* does not include typedef names for cv-qualified class types. See 10.1.3 [dcl.typedef] paragraph 4:

Argument and text removed from proposed resolution (October 2002):

10.1.3 [dcl.typedef] paragraph 5:

Here's a good question: in this example, should `X` be used as a name-for-linkage-purposes (FLP name)?

```
typedef class { } const X;
```

Because a *type-qualifier* is parsed as a *decl-specifier*, it isn't possible to declare cv-qualified and cv-unqualified typedefs for a type in a single declaration. Also, of course, there's no way to declare a typedef for the cv-unqualified version of a type for which only a cv-qualified version has a name. So, in the above example, if `X` isn't used as the FLP name, then there can be no FLP name. Also note that a FLP name usually represents a parameter type, where top-level cv-qualifiers are usually irrelevant anyway.

Data points: for the above example, Microsoft uses `X` as the FLP name; GNU and EDG do not.

My recommendation: for consistency with the direction we're going on this issue, for simplicity of description (e.g., "the first *class-name* declared by the declaration"), and for (very slightly) increased utility, I think Microsoft has this right.

> If the typedef declaration defines an unnamed class **type** (or enum **type**), the first *typedef-name* declared by the declaration to ~~be~~ **have** that ~~class~~ type ~~(or enum type)~~ **or a cv-qualified version thereof** is used to denote the class type (or enum type) for linkage purposes only (6.5 [basic.link]). [*Example:* ...

Proposed resolution (October 2002):

6.4.4 [basic.lookup.elab] paragraphs 2 and 3:

This sentence is deleted twice:

> ... ~~If this name lookup finds a *typedef-name*, the *elaborated-type-specifier* is ill-formed.~~ ...

Note that the above changes are included in N1376 as part of the resolution of issue 245.

_N4567_.5.1.1 [expr.prim.general] paragraph 7:

This is only a note, and it is at least incomplete (and quite possibly inaccurate), despite (or because of) its complexity. I propose to delete it.

> ... [*Note:* a *typedef-name* that names a class is a *class-name* (12.1 [class.name]). ~~Except as the *identifier* in the declarator for a constructor or destructor definition outside of a class *member-specification* (15.1 [class.ctor], 15.4 [class.dtor]), a *typedef-name* that names a class may be used in a *qualified-id* to refer to a constructor or destructor.~~]

10.1.3 [dcl.typedef] paragraph 4:

My first choice would have been to make this the primary statement about the equivalence of *typedef-name* and *class-name*, since the equivalence comes about as a result of a typedef declaration. Unfortunately, references to *class-name* point to 12.1 [class.name], so it would seem that the primary statement should be there instead. To avoid the possiblity of conflicts in the future, I propose to make this a note.

> **[*Note:*** A *typedef-name* that names a class **type, or a cv-qualified version thereof,** is **also** a *class-name* (12.1 [class.name]). If a *typedef-name* is used ~~following the *class-key* in an *elaborated-type-specifier* (10.1.7.3 [dcl.type.elab]), or in the *class-head* of a class declaration (12 [class]), or is used as the *identifier* in the declarator for a constructor or destructor declaration (15.1 [class.ctor], 15.4 [class.dtor]),~~ **to identify the subject of an *elaborated-type-specifier* (10.1.7.3 [dcl.type.elab]), class declaration (clause 12 [class]), constructor declaration (15.1 [class.ctor]), or destructor declaration (15.4 [class.dtor]),** the program is ill-formed. **]** [*Example:* ...

10.1.7.3 [dcl.type.elab] paragraph 2:

This is the only remaining (normative) statement that a *typedef-name* can't be used in an *elaborated-type-specifier*. The reference to template *type-parameter* is deleted by the resolution of issue 283.

> ... If the *identifier* resolves to a *typedef-name* ~~or a template *type-parameter*~~, the *elaborated-type-specifier* is ill-formed. [*Note:* ...

11 [dcl.decl] grammar rule *declarator-id*:

When I looked carefully into the statement of the rule prohibiting a *typedef-name* in a constructor declaration, it appeared to me that this grammar rule (inadvertently?) allows something that's always forbidden semantically.

declarator-id:
id-expression
::$_{opt}$ *nested-name-specifier*$_{opt}$ ~~*type-name*~~ ***class-name***

12.1 [class.name] paragraph 5:

Unlike the prohibitions against appearing in an *elaborated-type-specifier* or constructor or destructor declarator, each of which was expressed more than once, the prohibition against a *typedef-name* appearing in a *class-head* was previously stated only in 10.1.3 [dcl.typedef]. It seems to me that that prohibition belongs here instead. Also, it seems to me important to clarify that a *typedef-name* that is a *class-name* is still a *typedef-name*. Otherwise, the various prohibitions can be argued around easily, if perversely ("But that isn't a *typedef-name*, it's a *class-name*, it says so right there in 12.1 [class.name].")

> A *typedef-name* (10.1.3 [dcl.typedef]) that names a class **type or a cv-qualified version thereof** is **also** a *class-name*, but shall not be used ~~in an *elaborated-type-specifier*, see also 10.1.3 [dcl.typedef].~~ **as the *identifier* in a *class-head*.**

15.1 [class.ctor] paragraph 3:

The new nonterminal references are needed to really nail down what we're talking about here. Otherwise, I'm just eliminating redundancy. (A *typedef-name* that doesn't name a class type is no more valid here than one that does.)

> ~~A *typedef-name* that names a class is a *class-name* (10.1.3 [dcl.typedef]), however, a~~ **A** *typedef-name* ~~that names a class~~ shall not be used as the ~~*identifier*~~ ***class-name*** in the ~~declarator~~ ***declarator-id*** for a constructor declaration.

15.4 [class.dtor] paragraph 1:

The same comments apply here as to 15.1 [class.ctor].

> ... ~~A *typedef-name* that names a class is a *class-name* (7.1.3); however, a~~ **A** *typedef-name* ~~that names a class~~ shall not be used as the ~~*identifier*~~ ***class-name*** **following the** ~ in the declarator for a destructor declaration.

---

318. `struct A::A` **should not name the constructor of** `A`

Section: 6.4.3.1 [class.qual] **Status:** CD1 **Submitter:** John Spicer **Date:** 18 Oct 2001

[Voted into WP at April 2003 meeting.]

A use of an injected-class-name in an elaborated-type-specifier should not name the constructor of the class, but rather the class itself, because in that context we know that we're looking for a type. See issue 147.

Proposed Resolution (revised October 2002):

This clarifies the changes made in the TC for issue 147.

In 6.4.3.1 [class.qual] paragraph 1a replace:

> If the *nested-name-specifier* nominates a class C, and the name specified after the *nested-name-specifier*, when looked up in C, is the injected class name of C (clause 12 [class]), the name is instead considered to name the constructor of class C.

with

> In a lookup in which the constructor is an acceptable lookup result, if the *nested-name-specifier* nominates a class C and the name specified after the *nested-name-specifier*, when looked up in C, is the injected class name of C (clause 12 [class]), the name is instead considered to name the constructor of class C. [Note: For example, the constructor is not an acceptable lookup result in an elaborated type specifier so the constructor would not be used in place of the injected class name.]

Note that issue 263 updates a part of the same paragraph.

Append to the example:

```
struct A::A a2;  // object of type A
```

---

400. Using-declarations and the "struct hack"

Section: 6.4.3.2 [namespace.qual] **Status:** CD1 **Submitter:** Mark Mitchell **Date:** 22 Jan 2003

[Voted into WP at March 2004 meeting.]

Consider this code:

```
struct A { int i; struct i {}; };
struct B { int i; struct i {}; };
struct D : public A, public B { using A::i; void f (); };
void D::f () { struct i x; }
```

I can't find anything in the standard that says definitively what this means. 10.3.3 [namespace.udecl] says that a using-declaration shall name "a member of a base class" -- but here we have two members, the data member A::i and the class A::i.

Personally, I'd find it more attractive if this code did not work. I'd like "using A::i" to mean "lookup A::i in the usual way and bind B::i to that", which would mean that while "i = 3" would be valid in D::f, "struct i x" would not be. However, if there were no A::i data member, then "A::i" would find the struct and the code in D::f would be valid.

John Spicer: I agree with you, but unfortunately the standard committee did not.

I remembered that this was discussed by the committee and that a resolution was adopted that was different than what I hoped for, but I had a hard time finding definitive wording in the standard.

I went back though my records and found the paper that proposed a resolution and the associated committee motion that adopted the proposed resolution The paper is N0905, and "option 1" from that paper was adopted at the Stockholm meeting in July of 1996. The resolution is that "using A::i" brings in everything named i from A.

6.4.3.2 [namespace.qual] paragraph 2 was modified to implement this resolution, but interestingly that only covers the namespace case and not the class case. I think the class case was overlooked when the wording was drafted. A core issue should be opened to make sure the class case is handled properly.

Notes from April 2003 meeting:

This is related to issue 11. 10.3.3 [namespace.udecl] paragraph 10 has an example for namespaces.

Proposed resolution (October 2003):

Add a bullet to the end of 6.4.3.1 [class.qual] paragraph 1:

- the lookup for a name specified in a *using-declaration* (10.3.3 [namespace.udecl]) also finds class or enumeration names hidden within the same scope (6.3.10 [basic.scope.hiding]).

Change the beginning of 10.3.3 [namespace.udecl] paragraph 4 from

> A *using-declaration* used as a *member-declaration* shall refer to a member of a base class of the class being defined, shall refer to a member of an anonymous union that is a member of a base class of the class being defined, or shall refer to an enumerator for an enumeration type that is a member of a base class of the class being defined.

to

> In a *using-declaration* used as a *member-declaration*, the *nested-name-specifier* shall name a base class of the class being defined. Such a *using-declaration* introduces the set of declarations found by member name lookup (13.2 [class.member.lookup], 6.4.3.1 [class.qual]).

---

245. Name lookup in *elaborated-type-specifier*s

Section: 6.4.4 [basic.lookup.elab] **Status:** CD1 **Submitter:** Jack Rouse **Date:** 14 Sep 2000

[Voted into WP at April 2003 meeting.]

I have some concerns with the description of name lookup for elaborated type specifiers in 6.4.4 [basic.lookup.elab]:

1. Paragraph 2 has some parodoxical statements concerning looking up names that are simple identifers:

   > If the *elaborated-type-specifier* refers to an *enum-name* and this lookup does not find a previously declared *enum-name*, the *elaborated-type-specifier* is ill-formed. If the *elaborated-type-specifier* refers to an *[sic] class-name* and this lookup does not find a previously declared *class-name*... the *elaborated-type-specifier* is a declaration that introduces the *class-name* as described in 6.3.2 [basic.scope.pdecl]."

   It is not clear how an *elaborated-type-specifier* can refer to an *enum-name* or *class-name* given that the lookup does not find such a name and that *class-name* and *enum-name* are not part of the syntax of an *elaborated-type-specifier*.

2. The second sentence quoted above seems to suggest that the name found will not be used if it is not a class name. *typedef-name* names are ill-formed due to the sentence preceding the quote. If lookup finds, for instance, an *enum-name* then a new declaration will be created. This differs from C, and from the enum case, and can have surprising effects:

   ```
   struct S {
      enum E {
          one = 1
      };
      class E* p;     // declares a global class E?
   };
   ```

   Was this really the intent? If this is the case then some more work is needed on 6.4.4 [basic.lookup.elab]. Note that the section does not make finding a type template formal ill-formed, as is done in 10.1.7.3 [dcl.type.elab]. I don't see anything that makes a type template formal name a *class-name*. So the example in 10.1.7.3 [dcl.type.elab] of `friend class T;` where `T` is a template type formal would no longer be ill-formed with this interpretation because it would declare a new class `T`.

(See also issue 254.)

Notes from the 4/02 meeting:

This will be consolidated with the changes for issue 254. See also issue 298.

Proposed resolution (October 2002):

As given in N1376=02-0034. Note that the inserts and strikeouts in that document do not display correctly in all browsers; <del> --> <strike> and <ins> --> <b>, and the similar changes for the closing delimiters, seem to do the trick.

---

254. Definitional problems with *elaborated-type-specifier*s

Section: 6.4.4 [basic.lookup.elab] **Status:** CD1 **Submitter:** Clark Nelson **Date:** 26 Oct 2000

[Voted into WP at April 2003 meeting.]

1. The text in 6.4.4 [basic.lookup.elab] paragraph 2 twice refers to the possibility that an *elaborated-type-specifier* might have the form

   ```
   class-key identifier ;
   ```

   However, the grammar for *elaborated-type-specifier* does not include a semicolon.

2. In both 6.4.4 [basic.lookup.elab] and 10.1.7.3 [dcl.type.elab], the text asserts that an *elaborated-type-specifier* that refers to a *typedef-name* is ill-formed. However, it is permissible for the form of *elaborated-type-specifier* that begins with `typename` to refer to a *typedef-name*.

   This problem is the result of adding the `typename` form to the *elaborated-type-name* grammar without changing the verbiage correspondingly. It could be fixed either by updating the verbiage or by moving the `typename` syntax into its own production and referring to both nonterminals when needed.

(See also issue 180. If this issue is resolved in favor of a separate nonterminal in the grammar for the `typename` forms, the wording in that issue's resolution must be changed accordingly.)

Notes from 04/01 meeting:

The consensus was in favor of moving the `typename` forms out of the *elaborated-type-specifier* grammar.

Notes from the 4/02 meeting:

This will be consolidated with the changes for issue 245.

Proposed resolution (October 2002):

As given in N1376=02-0034.

---

141. Non-member function templates in member access expressions

Section: 6.4.5 [basic.lookup.classref] **Status:** CD1 **Submitter:** fvali **Date:** 31 July 1999

[Voted into the WP at the June, 2008 meeting.]

6.4.5 [basic.lookup.classref] paragraph 1 says,

> In a class member access expression (8.2.5 [expr.ref]), if the `.` or `->` token is immediately followed by an *identifier* followed by a `<`, the identifier must be looked up to determine whether the `<` is the beginning of a template argument list (17.2 [temp.names]) or a less-than operator. 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* and shall name a class or function template.

There do not seem to be any circumstances in which use of a non-member template function would be well-formed as the *id-expression* of a class member access expression.

Proposed Resolution (November, 2006):

Change 6.4.5 [basic.lookup.classref] paragraph 1 as follows:

> In a class member access expression (8.2.5 [expr.ref]), if the `.` or `->` token is immediately followed by an identifier followed by a `<`, the identifier must be looked up to determine whether the `<` is the beginning of a template argument list (17.2 [temp.names]) or a less-than operator. 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* and shall name a class ~~or function~~ template...

---

305. Name lookup in destructor call

Section: 6.4.5 [basic.lookup.classref] **Status:** CD1 **Submitter:** Mark Mitchell **Date:** 19 May 2001

[Voted into WP at the October, 2006 meeting.]

I believe this program is invalid:

```
struct A {
};

struct C {
  struct A {};
  void f ();
};

void C::f () {
  ::A *a;
  a->~A ();
}
```

The problem is that 6.4.5 [basic.lookup.classref] says that you have to look up `A` in both the context of the pointed-to-type (i.e., `::A`), and in the context of the postfix-expression (i.e., the body of `C::f`), and that if the name is found in both places it must name the same type in both places.

The EDG front end does not issue an error about this program, though.

Am I reading the standardese incorrectly?

John Spicer: I think you are reading it correctly. I think I've been hoping that this would get changed. Unlike other dual lookup contexts, this is one in which the compiler already knows the right answer (the type must match that of the left hand of the `->` operator). So I think that if either of the types found matches the one required, it should be sufficient. You can't say `a->~::A()`, which means you are forced to say `a->::A::~A()`, which disables the virtual mechanism. So you would have to do something like create a local typedef for the desired type.

See also issues 244, 399, and 466.

Proposed resolution (April, 2006):

1. Remove the indicated text from 6.4.5 [basic.lookup.classref] paragraph 2:

   > If the *id-expression* in a class member access (8.2.5 [expr.ref]) is an *unqualified-id*, and the type of the object expression is of a class type `C` ~~(or of pointer to a class type `C`)~~, the *unqualified-id* is looked up in the scope of class `C`...

2. Change 6.4.5 [basic.lookup.classref] paragraph 3 as indicated:

   > If the *unqualified-id* is `~`*type-name*, **the *type-name* is looked up in the context of the entire *postfix-expression*.** ~~and~~ **If** the type `T` of the object expression is of a class type `C` ~~(or of pointer to a class type `C`)~~, the *type-name* is **also** looked up ~~in the context of the entire *postfix-expression* and~~ in the scope of class `C`. ~~The *type-name* shall refer to a *class-name*. If *type-name* is found in both contexts, the name shall refer to the same class type. If the type of the object expression is of scalar type, the *type-name* is looked up in the scope of the complete *postfix-expression*.~~ **At least one of the lookups shall find a name that refers to (possibly cv-qualified) `T`. [*Example:***
   >
   > ```
   > struct A { };
   >
   > struct B {
   >   struct A { };
   >   void f(::A* a);
   > };
   >
   > void B::f(::A* a) {
   >   a->~A();  // OK, lookup in *a finds the injected-class-name
   > }
   > ```
   >
   > ***—end example*]**

[Note: this change also resolves issue 414.]

---

381. Incorrect example of base class member lookup

Section: 6.4.5 [basic.lookup.classref] **Status:** CD1 **Submitter:** Steve Adamczyk **Date:** 8 Nov 2002

[Voted into WP at October 2004 meeting.]

The example in 6.4.5 [basic.lookup.classref] paragraph 4 is wrong (see 14.2 [class.access.base] paragraph 5; the cast to the naming class can't be done) and needs to be corrected. This was noted when the final version of the algorithm for issue 39 was checked against it.

Proposed Resolution (October 2003):

Remove the entire note at the end of 6.4.5 [basic.lookup.classref] paragraph 4, including the entire example.

---

414. Multiple types found on destructor lookup

Section: 6.4.5 [basic.lookup.classref] **Status:** CD1 **Submitter:** John Spicer **Date:** 1 May 2003

[Voted into WP at the October, 2006 meeting.]

By 6.4.5 [basic.lookup.classref] paragraph 3, the following is ill-formed because the two lookups of the destructor name (in the scope of the class of the object and in the surrounding context) find different Xs:

```
struct X {};
int main() {
  X x;
  struct X {};
  x.~X();  // Error?
}
```

This is silly, because the compiler knows what the type has to be, and one of the things found matches that. The lookup should require only that one of the lookups finds the required class type.

Proposed resolution (April, 2005):

This issue is resolved by the resolution of issue 305.

---

216. Linkage of nameless class-scope enumeration types

Section: 6.5 [basic.link] **Status:** CD1 **Submitter:** Daveed Vandevoorde **Date:** 13 Mar 2000

[Moved to DR at 10/01 meeting.]

6.5 [basic.link] paragraph 4 says (among other things):

> A name having namespace scope has external linkage if it is the name of
>
> - [...]
> - a named enumeration (10.2 [dcl.enum]), or an unnamed enumeration defined in a typedef declaration in which the enumeration has the typedef name for linkage purposes (10.1.3 [dcl.typedef])

That prohibits for example:

```
typedef enum { e1 } *PE;
void f(PE) {}  // Cannot declare a function (with linkage) using a
               // type with no linkage.
```

However, the same prohibition was not made for class scope types. Indeed, 6.5 [basic.link] paragraph 5 says:

> In addition, a member function, static data member, class or enumeration of class scope has external linkage if the name of the class has external linkage.

That allows for:

```
struct S {
   typedef enum { e1 } *MPE;
   void mf(MPE) {}
};
```

My guess is that this is an unintentional consequence of 6.5 [basic.link] paragraph 5, but I would like confirmation on that.

Proposed resolution:

Change text in 6.5 [basic.link] paragraph 5 from:

> In addition, a member function, static data member, class or enumeration of class scope has external linkage if the name of the class has external linkage.

to:

> In addition, a member function, a static data member, a named class or enumeration of class scope, or an unnamed class or enumeration defined in a class-scope typedef declaration such that the class or enumeration has the typedef name for linkage purposes (10.1.3 [dcl.typedef]), has external linkage if the name of the class has external linkage.

---

319. Use of names without linkage in declaring entities with linkage

Section: 6.5 [basic.link] **Status:** CD1 **Submitter:** Clark Nelson **Date:** 29 Oct 2001

[Voted into WP at October 2004 meeting.]

According to 6.5 [basic.link] paragraph 8, "A name with no linkage ... shall not be used to declare an entity with linkage." This would appear to rule out code such as:

```
typedef struct {
  int i;
} *PT;
extern "C" void f(PT);
```

[likewise]

```
static enum { a } e;
```

which seems rather harmless to me.

See issue 132, which dealt with a closely related issue.

Andrei Iltchenko submitted the same issue via comp.std.c++ on 17 Dec 2001:

Paragraph 8 of Section 6.5 [basic.link] contains the following sentences: "A name with no linkage shall not be used to declare an entity with linkage. If a declaration uses a typedef name, it is the linkage of the type name to which the typedef refers that is considered."

The problem with this wording is that it doesn't cover cases where the type to which a typedef-name refers has no name. As a result it's not clear whether, for example, the following program is well-formed:

```
#include <vector>

int  main()
{
   enum  {   sz = 6u   };
   typedef int  (* aptr_type)[sz];
   typedef struct  data  {
      int   i,  j;
   }  * elem_type;
   std::vector<aptr_type>   vec1;
   std::vector<elem_type>   vec2;
}
```

Suggested resolution:

My feeling is that the rules for whether or not a typedef-name used in a declaration shall be treated as having or not having linkage ought to be modelled after those for dependent types, which are explained in 17.7.2.1 [temp.dep.type].

Add the following text at the end of Paragraph 8 of Section 6.5 [basic.link] and replace the following example:

> In case of the type referred to by a typedef declaration not having a name, the newly declared typedef-name has linkage if and only if its referred type comprises no names of no linkage excluding local names that are eligible for appearance in an integral constant-expression (8.20 [expr.const]). [Note: if the referred type contains a typedef-name that does not denote an unnamed class, the linkage of that name is established by the recursive application of this rule for the purposes of using typedef names in declarations.] [Example:
>
> ```
> void f()
> {
>    struct A { int x; };        // no linkage
>    extern A a;                 // ill-formed
>    typedef A B1
>    extern B b;                 // ill-formed
>
>    enum  {   sz = 6u   };
>    typedef int  (* C)[sz];     // C has linkage because sz can
>                                // appear in a constant expression
> }
> ```
>
> --end example.]

Additional issue (13 Jan 2002, from Andrei Iltchenko):

Paragraph 2 of Section 17.3.1 [temp.arg.type] is inaccurate and unnecessarily prohibits a few important cases; it says "A local type, a type with no linkage, an unnamed type or a type compounded from any of these types shall not be used as a template-argument for a template-parameter." The inaccuracy stems from the fact that it is not a type but its name that can have a linkage.

For example based on the current wording of 17.3.1 [temp.arg.type], the following example is ill-formed.

```
#include <vector>
struct  data  {
  int   i,  j;
};
int  main()
{
  enum  {   sz = 6u   };
  std::vector<int(*)[sz]>   vec1; // The types 'int(*)[sz]' and 'data*'
  std::vector<data*>        vec2; // have no names and are thus illegal
                                  // as template type arguments.
}
```

Suggested resolution:

Replace the whole second paragraph of Section 17.3.1 [temp.arg.type] with the following wording:

> A type whose name does not have a linkage or a type compounded from any such type shall not be used as a template-argument for a template-parameter. In case of a type `T` used as a template type argument not having a name, `T` constitutes a

valid template type argument if and only if the name of an invented typedef declaration referring to `T` would have linkage; see 3.5. [Example:

```
template <class T> class X { /* ... */ };
void f()
{
  struct S { /* ... */ };
  enum  {   sz = 6u   };

  X<S> x3;                    // error: a type name with no linkage
                              // used as template-argument
  X<S*> x4;                   // error: pointer to a type name with
                              // no linkage used as template-argument
  X<int(*)[sz]> x5;           // OK: since the name of typedef int
                              // (*pname)[sz] would have linkage
}
```

--end example] [Note: a template type argument may be an incomplete type (6.9 [basic.types]).]

Proposed resolution:

This is resolved by the changes for issue 389. The present issue was moved back to Review status in February 2004 because 389 was moved back to Review.

---

389. Unnamed types in entities with linkage

Section: 6.5 [basic.link] **Status:** CD1 **Submitter:** Daveed Vandevoorde **Date:** 31 Oct 2002

[Voted into WP at October 2004 meeting.]

6.5 [basic.link] paragraph 8 says (among other things):

> A name with no linkage (notably, the name of a class or enumeration declared in a local scope (6.3.3 [basic.scope.block])) shall not be used to declare an entity with linkage. If a declaration uses a typedef name, it is the linkage of the type name to which the typedef refers that is considered.

I would expect this to catch situations such as the following:

```
// File 1:
typedef struct {} *UP;
void f(UP) {}

// File 2:
typedef struct {} *UP; // Or: typedef struct {} U, *UP;
void f(UP);
```

The problem here is that most implementations must generate the same mangled name for "f" in two translation units. The quote from the standard above isn't quite clear, unfortunately: There is no type name to which the typedef refers.

A related situation is the following:

```
enum { no, yes } answer;
```

The variable "answer" is declared as having external linkage, but it is declared with an unnamed type. Section 6.5 [basic.link] talks about the linkage of *names*, however, and does therefore not prohibit this. There is no implementation issue for most compilers because they do not ordinarily mangle variable names, but I believe the intent was to allow that implementation technique.

Finally, these problems are much less relevant when declaring names with internal linkage. For example, I would expect there to be few problems with:

```
typedef struct {} *UP;
static void g(UP);
```

I recently tried to interpret 6.5 [basic.link] paragraph 8 with the assumption that types with no names have no linkage. Surprisingly, this resulted in many diagnostics on variable declarations (mostly like "answer" above).

I'm pretty sure the standard needs clarifying words in this matter, but which way should it go?

See also issue 319.

Notes from April 2003 meeting:

There was agreement that this check is not needed for variables and functions with extern "C" linkage, and a change there is desirable to allow use of legacy C headers. The check is also not needed for entities with internal linkage, but there was no strong sentiment for changing that case.

We also considered relaxing this requirement for extern "C++" variables but decided that we did not want to change that case.

We noted that if extern "C" functions are allowed an additional check is needed when such functions are used as arguments in calls of function templates. Deduction will put the type of the extern "C" function into the type of the template instance, i.e., there would be a need to mangle the name of an unnamed type. To plug that hole we need an additional requirement on the template created in such a case.

Proposed resolution (April 2003, revised slightly October 2003 and March 2004):

In 6.5 [basic.link] paragraph 8, change

> A name with no linkage (notably, the name of a class or enumeration declared in a local scope (6.3.3 [basic.scope.block])) shall not be used to declare an entity with linkage. If a declaration uses a typedef name, it is the linkage of the type name to which the typedef refers that is considered.

to

> A type is said to have linkage if and only if
>
> - it is a class or enumeration type that is named (or has a name for linkage purposes (10.1.3 [dcl.typedef])) and the name has linkage; or
> - it is a specialization of a class template (17 [temp]) [Footnote: a class template always has external linkage, and the requirements of 17.3.1 [temp.arg.type] and 17.3.2 [temp.arg.nontype] ensure that the template arguments will also have appropriate linkage]; or
> - it is a fundamental type (6.9.1 [basic.fundamental]); or
> - it is a compound type (6.9.2 [basic.compound]) other than a class or enumeration, compounded exclusively from types that have linkage; or
> - it is a cv-qualified (6.9.3 [basic.type.qualifier]) version of a type that has linkage.
>
> A type without linkage shall not be used as the type of a variable or function with linkage, unless the variable or function has extern "C" linkage (10.5 [dcl.link]). [*Note:* in other words, a type without linkage contains a class or enumeration that cannot be named outside of its translation unit. An entity with external linkage declared using such a type could not correspond to any other entity in another translation unit of the program and is thus not permitted. Also note that classes with linkage may contain members whose types do not have linkage, and that typedef names are ignored in the determination of whether a type has linkage.]

Change 17.3.1 [temp.arg.type] paragraph 2 from (note: this is the wording as updated by issue 62)

> The following types shall not be used as a *template-argument* for a template *type-parameter*:
>
> - a type whose name has no linkage
> - an unnamed class or enumeration type that has no name for linkage purposes (10.1.3 [dcl.typedef])
> - a cv-qualified version of one of the types in this list
> - a type created by application of declarator operators to one of the types in this list
> - a function type that uses one of the types in this list

to

> A type without linkage (6.5 [basic.link]) shall not be used as a *template-argument* for a template *type-parameter*.

Once this issue is ready, issue 319 should be moved back to ready as well.

---

474. Block-scope `extern` declarations in namespace members

Section: 6.5 [basic.link] **Status:** CD1 **Submitter:** Daveed Vandevoorde **Date:** 23 Jul 2004

[Voted into WP at October 2005 meeting.]

Consider the following bit of code:

```
namespace N {
  struct S {
    void f();
  };
}
using namespace N;
void S::f() {
  extern void g();  // ::g or N::g?
}
```

In 6.5 [basic.link] paragraph 7 the Standard says (among other things),

> When a block scope declaration of an entity with linkage is not found to refer to some other declaration, then that entity is a member of the innermost enclosing namespace.

The question then is whether `N` is an "enclosing namespace" for the local declaration of `g()`?

Proposed resolution (October 2004):

Add the following text as a new paragraph at the end of 10.3.1 [namespace.def]:

> The *enclosing namespaces* of a declaration are those namespaces in which the declaration lexically appears, except for a redeclaration of a namespace member outside its original namespace (e.g., a definition as specified in 10.3.1.2 [namespace.memdef]). Such a redeclaration has the same enclosing namespaces as the original declaration. [*Example:*

```
namespace Q {
  namespace V {
    void f(); // enclosing namespaces are the global namespace, Q, and Q::V
    class C { void m(); };
  }
  void V::f() { // enclosing namespaces are the global namespace, Q, and Q::V
    extern void h(); // ... so this declares Q::V::h
  }
  void V::C::m() { // enclosing namespaces are the global namespace, Q, and Q::V
  }
}
```

—*end example*]

---

270. Order of initialization of static data members of class templates

Section: 6.6.2 [basic.start.static] **Status:** CD1 **Submitter:** Jonathan H. Lundquist **Date:** 9 Feb 2001

[Moved to DR at 4/02 meeting.]

The Standard does not appear to address how the rules for order of initialization apply to static data members of class templates.

Suggested resolution: Add the following verbiage to either 6.6.2 [basic.start.static] or 12.2.3.2 [class.static.data]:

> Initialization of static data members of class templates shall be performed during the initialization of static data members for the first translation unit to have static initialization performed for which the template member has been instantiated. This requirement shall apply to both the static and dynamic phases of initialization.

Notes from 04/01 meeting:

Enforcing an order of initialization on static data members of class templates will result in substantial overhead on access to such variables. The problem is that the initialization be required as the result of instantiation in a function used in the initialization of a variable in another translation unit. In current systems, the order of initialization of static data data members of class templates is not predictable. The proposed resolution is to state that the order of initialization is undefined.

Proposed resolution (04/01, updated slightly 10/01):

Replace the following sentence in 6.6.2 [basic.start.static] paragraph 1:

> Objects with static storage duration defined in namespace scope in the same translation unit and dynamically initialized shall be initialized in the order in which their definition appears in the translation unit.

with

> Dynamic initialization of an object is either ordered or unordered. Explicit specializations and definitions of class template static data members have ordered initialization. Other class template static data member instances have unordered initialization. Other objects defined in namespace scope have ordered initialization. Objects defined within a single translation unit and with ordered initialization shall be initialized in the order of their definitions in the translation unit. The order of initialization is unspecified for objects with unordered initialization and for objects defined in different translation units.

Note that this wording is further updated by issue 362.

Note (07/01):

Brian McNamara argues against the proposed resolution. The following excerpt captures the central point of a long message on comp.std.c++:

> I have a class for representing linked lists which looks something like
>
> ```
> template <class T>
> class List {
>    ...  static List<T>* sentinel; ...
> };
>
> template <class T>
> List<T>* List<T>::sentinel( new List<T> ); // static member definition
> ```
>
> The sentinel list node is used to represent "nil" (the null pointer cannot be used with my implementation, for reasons which are immaterial to this discussion). All of the List's non-static member functions and constructors depend upon the value of the sentinel. Under the proposed resolution for issue #270, Lists cannot be safely instantiated before main() begins, as the sentinel's initialization is "unordered".
>
> (Some readers may propose that I should use the "singleton pattern" in the List class. This is undesirable, for reasons I shall describe at the end of this post at the location marked "[*]". For the moment, indulge me by assuming that "singleton" is not an adequate solution.)
>
> Though this is a particular example from my own experience, I believe it is representative of a general class of examples. It is common to use static data members of a class to represent the "distinguished values" which are important to instances of that class. It is imperative that these values be initialized before any instances of the class are created, as the instances depend on the values.

In a comp.std.c++ posting on 28 Jul 2001, Brian McNamara proposes the following alternative resolution:

Replace the following sentence in 6.6.2 [basic.start.static] paragraph 1:

> Objects with static storage duration defined in namespace scope in the same translation unit and dynamically initialized shall be initialized in the order in which their definition appears in the translation unit.

with

> Objects with static storage duration defined in namespace scope shall be initialized in the order described below.

and then after paragraph 1, add this text:

> Dynamic initialization is either ordered or quasi-ordered. Explicit specializations of class template static data members have ordered initialization. Other class template static data member instances have quasi-ordered initialization. All other objects defined in namespace scope have ordered initialization. The order of initialization is specified as follows:
>
> - Objects that are defined within a single translation unit and that have ordered initialization shall be initialized in the order of their definitions in the translation unit.
> - Objects that are defined only within a single translation unit and that have quasi-ordered initialization shall also be initialized in the order of their definitions in the translation unit -- that is, as though these objects had ordered initialization.
> - Objects that are defined within multiple translation units (which, therefore, must have quasi-ordered initialization) shall be initialized as follows: in exactly one translation unit (*which* one is unspecified), the object shall be treated as though it has ordered initialization; in the other translation units which define the object, the object will be initialized before all other objects that have ordered initialization in those translation units.
> - For any two objects, "X" and "Y", with static storage duration and defined in namespace scope, if the previous bullets do not imply a relationship for the initialization ordering between "X" and "Y", then the relative initialization order of these objects is unspecified.

along with a non-normative note along the lines of

> [Note: The intention is that translation units can each be compiled separately with no knowledge of what objects may be re-defined in other translation units. Each translation unit can contain a method which initializes all objects (both quasi-ordered and ordered) as though they were ordered. When these translation units are linked together to create an executable program, all of these objects can be initialized by simply calling the initialization methods (one from each translation unit) in any order. Quasi-ordered objects require some kind of guard to ensure that they are not initialized more than once (the first attempt to initialize such an object should succeed; any subsequent attempts should simply be ignored).]

Erwin Unruh replies: There is a point which is not mentioned with this posting. It is the cost for implementing the scheme. It requires that each static template variable is instantiated in ALL translation units where it is used. There has to be a flag for each of these variables and this flag has to be checked in each TU where the instantiation took place.

I would reject this idea and stand with the proposed resolution of issue 270.

There just is no portable way to ensure the "right" ordering of construction.

Notes from 10/01 meeting:

The Core Working Group reaffirmed its previous decision.

---

441. Ordering of static reference initialization

Section: 6.6.2 [basic.start.static] **Status:** CD1 **Submitter:** Mike Miller **Date:** 1 Dec 2003

[Voted into WP at April 2005 meeting.]

I have a couple of questions about 6.6.2 [basic.start.static], "Initialization of non-local objects." I believe I recall some discussion of related topics, but I can't find anything relevant in the issues list.

The first question arose when I discovered that different implementations treat reference initialization differently. Consider, for example, the following (namespace-scope) code:

```
int i;
int& ir = i;
int* ip = &i;
```

Both initializers, "i" and "&i", are constant expressions, per 8.20 [expr.const] paragraph 4-5 (a reference constant expression and an address constant expression, respectively). Thus, both initializations are categorized as static initialization, according to 6.6.2 [basic.start.static] paragraph 1:

> Zero-initialization and initialization with a constant expression are collectively called static initialization; all other initialization is dynamic initialization.

However, that does not mean that both ir and ip must be initialized at the same time:

> Objects of POD types (3.9) with static storage duration initialized with constant expressions (5.19) shall be initialized before any dynamic initialization takes place.

Because "int&" is not a POD type, there is no requirement that it be initialized before dynamic initialization is performed, and implementations differ in this regard. Using a function called during dynamic initialization to print the values of "ip" and "&ir", I found that g++, Sun, HP, and Intel compilers initialize ir before dynamic initialization and the Microsoft compiler does not. All initialize ip before dynamic initialization. I believe this is conforming (albeit inconvenient :-) behavior.

So, my first question is whether it is intentional that a reference of static duration, initialized with a reference constant expression, need not be initialized before dynamic initialization takes place, and if so, why?

The second question is somewhat broader. As 6.6.2 [basic.start.static] is currently worded, it appears that there are no requirements on when ir is initialized. In fact, there is a whole category of objects -- non-POD objects initialized with a constant expression -- for which no ordering is specified. Because they are categorized as part of "static initialization," they are not subject to the requirement that they "shall be initialized in the order in which their definition appears in the translation unit." Because they are not POD types, they are not required to be initialized before dynamic initialization occurs. Am I reading this right?

My preference would be to change 6.6.2 [basic.start.static] paragraph 1 so that 1) references are treated like POD objects with respect to initialization, and 2) "static initialization" applies only to POD objects and references. Here's some sample wording to illustrate:

Suggested resolution:

> Objects with static storage duration (3.7.1) shall be zero-initialized (8.5) before any other initialization takes place. Initializing a reference, or an object of POD type, of static storage duration with a constant expression (5.19) is called constant initialization. Together, zero-initialization and constant initialization are called static initialization; all other initialization is dynamic initialization. Static initialization shall be performed before any dynamic initialization takes place. [Remainder unchanged.]

Proposed Resolution:

Change 6.6.2 [basic.start.static] paragraph 1 as follows:

> Objects with static storage duration (3.7.1) shall be zero-initialized (8.5) before any other initialization takes place. **Initializing a reference, or an object of POD type, of static storage duration with a constant expression (5.19) is called *constant initialization*. Together, zero-initialization and constant initialization are** ~~Zero-initialization and initialization with a constant expression are collectively~~ called *static initialization*; all other initialization is *dynamic initialization*. **Static initialization shall be performed** ~~Objects of POD types (3.9) with static storage duration initialized with constant expressions (5.19) shall be initialized~~ before any dynamic initialization takes place.

---

688. Constexpr constructors and static initialization

Section: 6.6.2 [basic.start.static] **Status:** CD1 **Submitter:** Peter Dimov **Date:** 26 March, 2008

[Voted into the WP at the September, 2008 meeting (resolution in paper N2757).]

Given this literal type,

```
struct X {
    constexpr X() { }
};
```

and this definition,

```
static X x;
```

the current specification does not require that `x` be statically initialized because it is not "initialized with a constant expression" (6.6.1 [basic.start.main] paragraph 1).

Lawrence Crowl:

This guarantee is essential for atomics.

Jens Maurer:

Suggestion:

> A reference with static storage duration or an object of literal type with static storage duration can be initialized with a constant expression (8.20 [expr.const]) or with a constexpr constructor; this is called constant initialization.

(Not spelling out "default constructor" makes it easier to handle multiple-parameter constexpr constructors, where there isn't "a" constant expression but several.)

Peter Dimov:

In addition, there is a need to enforce static initialization for non-literal types: `std::shared_ptr`, `std::once_flag`, and `std::atomic_*` all have nontrivial copy constructors, making them non-literal types. However, we need a way to ensure that a constexpr constructor called with constant expressions will guarantee static initialization, regardless of the nontriviality of the copy constructor.

Proposed resolution (April, 2008):

1. Change 6.6.2 [basic.start.static] paragraph 1 as follows:

> ...~~A reference with static storage duration and an object of trivial or literal type with static storage duration can be initialized with a constant expression (8.20 [expr.const]); this~~ **If a reference with static storage duration is initialized with a constant expression (8.20 [expr.const]) or if the initialization of an object with static storage duration satisfies the requirements for the object being declared with** `constexpr` **(10.1.5 [dcl.constexpr]), that initialization** is called *constant initialization*...

2. Change 9.7 [stmt.dcl] paragraph 4 as follows:

> ...~~A local object of trivial or literal type (6.9 [basic.types]) with static storage duration initialized with *constant-expressions* is initialized~~ **Constant initialization (6.6.2 [basic.start.static]) of a local entity with static storage duration is performed** before its block is first entered...

3. Change 10.1.5 [dcl.constexpr] paragraph 7 as follows:

> A `constexpr` specifier used in an object declaration declares the object as `const`. Such an object shall be initialized, and every expression that appears in its initializer (11.6 [dcl.init]) shall be a constant expression. Every implicit conversion used in converting the initializer expressions **and every constructor call used for the initialization** shall be one of those allowed in a constant expression (8.20 [expr.const])...

4. Replace 11.6.1 [dcl.init.aggr] paragraph 14 as follows:

> ~~When an aggregate with static storage duration is initialized with a brace-enclosed *initializer-list*, if all the member initializer expressions are constant expressions, and the aggregate is a trivial type, the initialization shall be done during the static phase of initialization (6.6.2 [basic.start.static]); otherwise, it is unspecified whether the initialization of members with constant expressions takes place during the static phase or during the dynamic phase of initialization.~~ **[*Note:* The order of initialization for aggregates with static storage duration is specified in 6.6.2 [basic.start.static] and 9.7 [stmt.dcl]. —*end note*]**

(Note: the change to 6.6.2 [basic.start.static] paragraph 1 needs to be reconciled with the conflicting change in issue 684.)

---

28. 'exit', 'signal' and static object destruction

Section: 6.6.3 [basic.start.dynamic] **Status:** CD1 **Submitter:** Martin J. O'Riordan **Date:** 19 Oct 1997

[Voted into the WP at the June, 2008 meeting.]

The C++ standard has inherited the definition of the 'exit' function more or less unchanged from ISO C.

However, when the 'exit' function is called, objects of static extent which have been initialised, will be destructed if their types posses a destructor.

In addition, the C++ standard has inherited the definition of the 'signal' function and its handlers from ISO C, also pretty much unchanged.

The C standard says that the only standard library functions that may be called while a signal handler is executing, are the functions 'abort', 'signal' and 'exit'.

This introduces a bit of a nasty turn, as it is not at all unusual for the destruction of static objects to have fairly complex destruction semantics, often associated with resource release. These quite commonly involve apparently simple actions such as calling 'fclose' for a FILE handle.

Having observed some very strange behaviour in a program recently which in handling a SIGTERM signal, called the 'exit' function as indicated by the C standard.

But unknown to the programmer, a library static object performed some complicated resource deallocation activities, and the program crashed.

The C++ standard says nothing about the interaction between signals, exit and static objects. My observations, was that in effect, because the destructor called a standard library function other than 'abort', 'exit' or 'signal', while transitively in the execution context of the signal handler, it was in fact non-compliant, and the behaviour was undefined anyway.

This is I believe a plausible judgement, but given the prevalence of this common programming technique, it seems to me that we need to say something a lot more positive about this interaction.

Curiously enough, the C standard fails to say anything about the analogous interaction with functions registered with 'atexit' ;-)

Proposed Resolution (10/98):

The current Committee Draft of the next version of the ISO C standard specifies that the only standard library function that may be called while a signal handler is executing is 'abort'. This would solve the above problem.

[This issue should remain open until it has been decided that the next version of the C++ standard will use the next version of the C standard as the basis for the behavior of 'signal'.]

Notes (November, 2006):

C89 is slightly contradictory here: It allows any signal handler to *terminate* by calling `abort`, `exit`, `longjmp`, but (for asynchronous signals, i.e. not those produced by `abort` or `raise`) then makes calling any library function other than `signal` with the current signal undefined behavior (C89 7.7.1.1). For synchronous signals, C99 forbids calls to `raise`, but imposes no other restrictions. For asynchronous signals, C99 allows only calls to `abort`, `_Exit`, and `signal` with the current signal (C99 7.14.1.1). The current C++ WP refers to "plain old functions" and "conforming C programs" (21.10 [support.runtime] paragraph 6).

Proposed Resolution (November, 2006):

Change the footnote in 21.10 [support.runtime] paragraph 6 as follows:

> In particular, a signal handler using exception handling is very likely to have problems. **Also, invoking `std::exit` may cause destruction of objects, including those of the standard library implementation, which, in general, yields undefined behavior in a signal handler (see 4.6 [intro.execution]).**

---

521. Requirements for exceptions thrown by allocation functions

Section: 6.7.4.1 [basic.stc.dynamic.allocation] **Status:** CD1 **Submitter:** Alisdair Meredith **Date:** 22 May 2005

[Voted into WP at the October, 2006 meeting.]

According to 6.7.4.1 [basic.stc.dynamic.allocation] paragraph 3,

> Any other allocation function that fails to allocate storage shall only indicate failure by throwing an exception of class `std::bad_alloc` (21.6.3.1 [bad.alloc]) or a class derived from `std::bad_alloc`.

Shouldn't this statement have the usual requirements for an unambiguous and accessible base class?

Proposed resolution (April, 2006):

Change the last sentence of 6.7.4.1 [basic.stc.dynamic.allocation] paragraph 3 as indicated:

> Any other allocation function that fails to allocate storage shall ~~only~~ indicate failure **only** by throwing an exception of ~~class `std::bad_alloc` (21.6.3.1 [bad.alloc]) or a class derived from `std::bad_alloc`~~ **a type that would match a handler (18.3 [except.handle]) of type `std::bad_alloc` (21.6.3.1 [bad.alloc])**.

---

220. All deallocation functions should be required not to throw

Section: 6.7.4.2 [basic.stc.dynamic.deallocation] **Status:** CD1 **Submitter:** Herb Sutter **Date:** 31 Mar 2000

[Voted into the WP at the September, 2008 meeting (resolution in paper N2757).]

[Picked up by evolution group at October 2002 meeting.]

The default global operators `delete` are specified to not throw, but there is no requirement that replacement global, or class-specific, operators `delete` must not throw. That ought to be required.

In particular:

- 6.7.4.2 [basic.stc.dynamic.deallocation] paragraph 2, at the end of the first sentence, should also require that no exceptions be thrown.
- 15.5 [class.free], including its examples, should require nonthrowing class-specific `operator delete`.
- 20.5.4.6 [replacement.functions] paragraph 2 should append `throw()` to the signature of each of the four operators `delete`.

We already require that all versions of an allocator's deallocate() must not throw, so that part is okay.

Rationale (04/00):

1. Replacement deallocation functions are already required not to throw an exception (cf 20.5.4.8 [res.on.functions] paragraph 2, as applied to 21.6.2.1 [new.delete.single] paragraph 12 and 21.6.2.2 [new.delete.array] paragraph 11).
2. Section 20.5.4.6 [replacement.functions] is describing the signatures of the functions to be replaced; exception specfications are not part of the signature.
3. There does not appear to be any pressing need to require that class member deallocation functions not throw.

Note (March, 2008):

The Evolution Working Group has accepted the intent of this issue and referred it to CWG for action for C++0x (see paper J16/07-0033 = WG21 N2173).

Proposed resolution (March, 2008):

Change 6.7.4.2 [basic.stc.dynamic.deallocation] paragraph 3 as follows:

A deallocation function shall not terminate by throwing an exception. The value of the first argument supplied to a deallocation function...

---

348. `delete` **and user-written deallocation functions**

Section: 6.7.4.2 [basic.stc.dynamic.deallocation] **Status:** CD1 **Submitter:** Ruslan Abdikeev **Date:** 1 April 2002

[Voted into WP at October 2005 meeting.]

Standard is clear on behaviour of default allocation/deallocation functions. However, it is surpisingly vague on requirements to the behaviour of user-defined deallocation function and an interaction between delete-expression and deallocation function. This caused a heated argument on fido7.su.c-cpp newsgroup.

Resume:

It is not clear if user-supplied deallocation function is called from delete-expr when the operand of delete-expr is the null pointer (8.3.5 [expr.delete]). If it is, standard does not specify what user-supplied deallocation function shall do with the null pointer operand (21.6.2 [new.delete]). Instead, Standard uses the term "has no effect", which meaning is too vague in context given (8.3.5 [expr.delete]).

Description:

Consider statements

```
char* p= 0; //result of failed non-throwing ::new char[]
::delete[] p;
```

Argument passed to delete-expression is valid - it is the result of a call to the non-throwing version of ::new, which has been failed. 8.3.5 [expr.delete] paragraph 1 explicitly prohibit us to pass 0 without having the ::new failure.

Standard does NOT specify whether user-defined deallocation function should be called in this case, or not.

Specifically, standard says in 8.3.5 [expr.delete] paragraph 2:

> ...if the value of the operand of delete is the null pointer the operation has no effect.

Standard doesn't specify term "has no effect". It is not clear from this context, whether the called deallocation function is required to have no effect, or delete-expression shall not call the deallocation function.

Furthermore, in para 4 standard says on default deallocation function:

> If the delete-expression calls the implementation deallocation function (6.7.4.2 [basic.stc.dynamic.deallocation]), if the operand of the delete expression is not the null pointer constant, ...

Why it is so specific on interaction of default deallocation function and delete-expr?

If "has no effect" is a requirement to the deallocation function, then it should be stated in 6.7.4.2 [basic.stc.dynamic.deallocation], or in 21.6.2.1 [new.delete.single] and 21.6.2.2 [new.delete.array], and it should be stated explicitly.

Furthermore, standard does NOT specify what actions shall be performed by user-supplied deallocation function if NULL is given (21.6.2.1 [new.delete.single] paragraph 12):

> Required behaviour: accept a value of ptr that is null or that was returned by an earlier call to the default `operator new(std::size_t)` or `operator new(std::size_t, const std::nothrow_t&)`.

The same corresponds to ::delete[] case.

Expected solution:

1. Make it clear that delete-expr will not call deallocation function if null pointer is given (in 8.3.5 [expr.delete]).
2. Specify what user deallocation function shall do when null is given (either in 6.7.4.2 [basic.stc.dynamic.deallocation], or in 21.6.2.1 [new.delete.single], and 21.6.2.2 [new.delete.array]).

Notes from October 2002 meeting:

We believe that study of 21.6.2.1 [new.delete.single] paragraphs 12 and 13, 21.6.2.2 [new.delete.array] paragraphs 11 and 12, and 6.7.4.2 [basic.stc.dynamic.deallocation] paragraph 3 shows that the system-provided operator delete functions must accept a null pointer and ignore it. Those sections also show that a user-written replacement for the system-provided operator delete functions must accept a null pointer. There is no requirement that such functions ignore a null pointer, which is okay -- perhaps the reason for replacing the system-provided functions is to do something special with null pointer values (e.g., log such calls and return).

We believe that the standard should not require an implementation to call a delete function with a null pointer, but it must allow that. For the system-provided delete functions or replacements thereof, the standard already makes it clear that the delete function must accept a null pointer. For class-specific delete functions, we believe the standard should require that such functions accept a null pointer, though it should not mandate what they do with null pointers.

8.3.5 [expr.delete] needs to be updated to say that it is unspecified whether or not the operator delete function is called with a null pointer, and 6.7.4.2 [basic.stc.dynamic.deallocation] needs to be updated to say that any deallocation function must accept a null pointer.

Proposed resolution (October, 2004):

1. Change 8.3.5 [expr.delete] paragraph 2 as indicated:

   > If the operand has a class type, the operand is converted to a pointer type by calling the above-mentioned conversion function, and the converted operand is used in place of the original operand for the remainder of this section. In either alternative, ~~if~~ the value of the operand of `delete` ~~is the null pointer the operation has no effect~~ **may be a null pointer value. If it is not a null pointer value, in** ~~In~~ the first alternative (*delete object*), the value of the operand of `delete` shall be a pointer to a non-array object or a pointer to a sub-object (4.5 [intro.object]) representing a base class of such an object (clause 13 [class.derived])...

2. Change 8.3.5 [expr.delete] paragraph 4 as follows (note that the old wording reflects the changes proposed by issue 442:

   > The *cast-expression* in a *delete-expression* shall be evaluated exactly once. ~~If the delete-expression calls the implementation deallocation function (6.7.4.2 [basic.stc.dynamic.deallocation]), and if the value of the operand of the delete-expression is not a null pointer, the deallocation function will deallocate the storage referenced by the pointer thus rendering the pointer invalid. [*Note:* the value of a pointer that refers to deallocated storage is indeterminate. —*end note*]~~

3. Change 8.3.5 [expr.delete] paragraphs 6-7 as follows:

   > ~~The~~ **If the value of the operand of the *delete-expression* is not a null pointer value, the** *delete-expression* will invoke the destructor (if any) for the object or the elements of the array being deleted. In the case of an array, the elements will be destroyed in order of decreasing address (that is, in reverse order of the completion of their constructor; see 15.6.2 [class.base.init]).
   >
   > ~~The~~ **If the value of the operand of the *delete-expression* is not a null pointer value, the** *delete-expression* will call a deallocation function (6.7.4.2 [basic.stc.dynamic.deallocation]). **Otherwise, it is unspecified whether the deallocation function will be called.** [*Note:* The deallocation function is called regardless of whether the destructor for the object or some element of the array throws an exception. —*end note*]

4. Change 6.7.4.2 [basic.stc.dynamic.deallocation] paragraph 3 as indicated:

   > The value of the first argument supplied to ~~one of the~~ **a** deallocation function~~s provided in the standard library~~ may be a null pointer value; if so, **and if the deallocation function is one supplied in the standard library,** the call ~~to the deallocation function~~ has no effect. Otherwise, the value supplied to `operator delete(void*)` in the standard library shall be one of the values returned by a previous invocation of either `operator new(std::size_t)` or `operator new(std::size_t, const std::nothrow_t&)` in the standard library, and the value supplied to `operator delete[](void*)` in the standard library shall be one of the values returned by a previous invocation of either `operator new[](std::size_t)` or `operator new[](std::size_t, const std::nothrow_t&)` in the standard library.

[Note: this resolution also resolves issue 442.]

---

119. Object lifetime and aggregate initialization

Section: 6.8 [basic.life] **Status:** CD1 **Submitter:** Jack Rouse **Date:** 20 May 1999

[Moved to DR at 4/02 meeting.]

Jack Rouse: 6.8 [basic.life] paragraph 1 includes:

> The lifetime of an object is a runtime property of the object. The lifetime of an object of type `T` begins when:
>
> - storage with the proper alignment and size for type `T` is obtained, and
> - if `T` is a class type with a non-trivial constructor (15.1 [class.ctor]), the constructor call has completed.

Consider the code:

```
struct B {
    B( int = 0 );
    ~B();
};

struct S {
    B b1;
};

int main()
{
    S s = { 1 };
    return 0;
}
```

In the code above, class `S` does have a non-trivial constructor, the default constructor generated by the compiler. According the text above, the lifetime of the auto `s` would never begin because a constructor for `S` is never called. I think the second case in the text needs to include aggregate initialization.

Mike Miller: I see a couple of ways of fixing the problem. One way would be to change "the constructor call has completed" to "the object's initialization is complete."

Another would be to add following "a class type with a non-trivial constructor" the phrase "that is not initialized with the brace notation (11.6.1 [dcl.init.aggr])."

The first formulation treats aggregate initialization like a constructor call; even POD-type members of an aggregate could not be accessed before the aggregate initialization completed. The second is less restrictive; the POD-type members of the aggregate would be usable before the initialization, and the members with non-trivial constructors (the only way an aggregate can acquire a non-trivial constructor) would be protected by recursive application of the lifetime rule.

Proposed resolution (04/01):

In 6.8 [basic.life] paragraph 1, change

> If `T` is a class type with a non-trivial constructor (15.1 [class.ctor]), the constructor call has completed.

to

> If `T` is a class type with a non-trivial constructor (15.1 [class.ctor]), the initialization is complete. [*Note:* the initialization can be performed by a constructor call or, in the case of an aggregate with an implicitly-declared non-trivial default constructor, an aggregate initialization (11.6.1 [dcl.init.aggr]).]

---

274. Cv-qualification and char-alias access to out-of-lifetime objects

Section: 6.8 [basic.life] **Status:** CD1 **Submitter:** Mike Miller **Date:** 14 Mar 2001

[Voted into WP at April 2003 meeting.]

The wording in 6.8 [basic.life] paragraph 6 allows an lvalue designating an out-of-lifetime object to be used as the operand of a `static_cast` only if the conversion is ultimately to "`char&`" or "`unsigned char&`". This description excludes the possibility of using a cv-qualified version of these types for no apparent reason.

Notes on 04/01 meeting:

The wording should be changed to allow cv-qualified char types.

Proposed resolution (04/01):

In 6.8 [basic.life] paragraph 6 change the third bullet:

- the lvalue is used as the operand of a `static_cast` (8.2.9 [expr.static.cast]) (except when the conversion is ultimately to `char&` or `unsigned char&`), or

to read:

- the lvalue is used as the operand of a `static_cast` (8.2.9 [expr.static.cast]) except when the conversion is ultimately to *cv* `char&` or *cv* `unsigned char&`, or

---

404. Unclear reference to construction with non-trivial constructor

Section: 6.8 [basic.life] **Status:** CD1 **Submitter:** Mike Miller **Date:** 8 Apr 2003

[Voted into WP at March 2004 meeting.]

6.8 [basic.life] paragraph 1 second bullet says:

> if T is a class type with a non-trivial constructor (12.1), the constructor call has completed.

This is confusing; what was intended is probably something like

> if T is a class type and the constructor invoked to create the object is non-trivial (12.1), the constructor call has completed.

Proposed Resolution (October 2003):

As given above.

---

594. Coordinating issues 119 and 404 with delegating constructors

Section: 6.8 [basic.life] **Status:** CD1 **Submitter:** Tom Plum **Date:** 30 August 2006

[Voted into the WP at the September, 2008 meeting.]

In ISO/IEC 14882:2003, the second bullet of 6.8 [basic.life] paragraph 1 reads,

> if `T` is a class type with a non-trivial constructor (15.1 [class.ctor]), the constructor call has completed.

Issue 119 pointed out that aggregate initialization can be used with some classes with a non-trivial implicitly-declared default constructor, and that in such cases there is no call to the object's constructor. The resolution for that issue was to change the previously-cited wording to read,

> If `T` is a class type with a non-trivial constructor (15.1 [class.ctor], **the initialization is complete**.

Later (but before the WP was revised with the wording from the resolution of issue 119), issue 404 changed the 2003 wording to read,

> If `T` is a class type **and the constructor invoked to create the object is non-trivial** (15.1 [class.ctor]), the constructor call has completed.

thus reversing the effect of issue 119, whose whole purpose was to cover objects with non-trivial constructors that are *not* invoked.

Through an editorial error, the post-Redmond draft (N1905) still contained the original 2003 wording that should have been replaced by the resolution of issue 119, *in addition to* the new wording from the resolution:

> if `T` is a class type and the constructor invoked to create the object is non-trivial (15.1 [class.ctor]), the constructor call has completed. the initialization is complete.

Finally, during the application of the edits for delegating constructors (N1986), this editing error was "fixed" by retaining the original 2003 wording (which was needed for the application of the change specified in N1986), so that the current draft (N2009) reads,

> if `T` is a class type and the constructor invoked to create the object is non-trivial (15.1 [class.ctor]), the principal constructor call 15.6.2 [class.base.init]) has completed.

Because the completion of the call to the principal constructor corresponds to the point at which the object is "fully constructed" (18.2 [except.ctor] paragraph 2), i.e., its initialization is complete, I believe that the exact wording of the issue 119 resolution would be correct and should be restored verbatim.

Proposed resolution (June, 2008):

Change 6.8 [basic.life] paragraph 1 as follows:

> The *lifetime* of an object is a runtime property of the object. **An object is said to have non-trivial initialization if it is of a class or aggregate type and it or one of its members is initialized by a constructor other than a trivial default constructor. [*Note:* Initialization by a trivial copy constructor is non-trivial initialization. —*end note*]** The lifetime of an object ~~of type `T`~~ begins when:
>
> - storage with the proper alignment and size ~~for type `T`~~ is obtained, and
> - ~~if `T` is a class type and the constructor invoked to create the object is non-trivial (15.1 [class.ctor]), the principal constructor call (15.6.2 [class.base.init]) has completed. [*Note:* the initialization can be performed by a constructor call or, in the case of an aggregate with an implicitly-declared non-trivial default constructor, an aggregate initialization 11.6.1 [dcl.init.aggr]. —*end note*]~~ **the object has non-trivial initialization, its initialization is complete.**
>
> The lifetime of an object of type `T` ends when...

---

644. Should a trivial class type be a literal type?

Section: 6.9 [basic.types] **Status:** CD1 **Submitter:** Alisdair Meredith **Date:** 8 Aug 2007

[Voted into the WP at the June, 2008 meeting.]

The original proposed wording for 6.9 [basic.types] paragraph 11 required a constexpr constructor for a literal class only "if the class has at least one user-declared constructor." This wording was dropped during the review by CWG out of a desire to ensure that literal types not have any uninitialized members. Thus, a class like

```
struct pixel {
    int x, y;
};
```

is not a literal type. However, if an object of that type is aggregate-initialized or value-initialized, there can be no uninitialized members; the missing wording should be restored in order to permit use of expressions like `pixel().x` as constant expressions.

Proposed resolution (February, 2008):

Change 6.9 [basic.types] paragraph 10 as follows:

> A type is a *literal type* if it is:
>
> - a scalar type; or
> - a class type (clause 12 [class]) with
>   - a trivial copy constructor,
>   - a trivial destructor,

    - **a trivial default constructor or** at least one constexpr constructor other than the copy constructor,
    - ~~no virtual base classes, and~~
    - all non-static data members and base classes of literal types; or
- an array of literal type.

---

158. Aliasing and qualification conversions

Section: 6.10 [basic.lval] **Status:** CD1 **Submitter:** Mike Stump **Date:** 20 Aug 1999

[Moved to DR at 4/02 meeting.]

6.10 [basic.lval] paragraph 15 lists the types via which an lvalue can be used to access the stored value of an object; using an lvalue type that is not listed results in undefined behavior. It is permitted to add cv-qualification to the actual type of the object in this access, but only at the top level of the type ("a cv-qualified version of the dynamic type of the object").

However, 7.5 [conv.qual] paragraph 4 permits a "conversion [to] add cv-qualifiers at levels other than the first in multi-level pointers." The combination of these two rules allows creation of pointers that cannot be dereferenced without causing undefined behavior. For instance:

```
int* jp;
const int * const * pl = &jp;
*pl;    // undefined behavior!
```

The reason that `*pl` results in undefined behavior is that the type of the lvalue is `const int * const*`, which is *not* "a cv-qualified version of" `int*`.

Since the conversion is permitted, we must give it defined semantics, hence we need to fix the wording in 6.10 [basic.lval] to include all possible conversions of the type via 7.5 [conv.qual].

Proposed resolution (04/01):

Add a new bullet to 6.10 [basic.lval] paragraph 15, following "a cv-qualified version of the dynamic type of the object:"

- A type similar (as defined in 7.5 [conv.qual]) to the dynamic type of the object,

---

649. Optionally ill-formed extended alignment requests

Section: 6.11 [basic.align] **Status:** CD1 **Submitter:** Mike Miller **Date:** 12 Aug 2007

[Voted into the WP at the September, 2008 meeting.]

The requirements on an implementation when presented with an *alignment-specifier* not supported by that implementation in that context are contradictory: 6.11 [basic.align] paragraph 9 says,

> If a request for a specific extended alignment in a specific context is not supported by an implementation, the implementation may reject the request as ill-formed. The implementation may also silently ignore the requested alignment.

In contrast, 10.6.2 [dcl.align] paragraph 2, bullet 4 says simply,

- if the constant expression evaluates to an extended alignment and the implementation does not support that alignment in the context of the declaration, the program is ill-formed

with no provision to "silently ignore" the requested alignment. These two passages need to be reconciled.

If the outcome of the reconciliation is to grant implementations the license to accept and ignore extended alignment requests, the specification should be framed in terms of mechanisms that already exist in the Standard, such as undefined behavior and/or conditionally-supported constructs; "ill-formed" is a category that is defined by the Standard, not something that an implementation can decide.

Notes from the February, 2008 meeting:

The consensus was that such requests should be ill-formed and require a diagnostic. However, it was also observed that an implementation need not reject an ill-formed program; the only requirement is that it issue a diagnostic. It would thus be permissible for an implementation to "noisily ignore" (as opposed to "silently ignoring") an unsupported alignment request.

Proposed resolution (June, 2008):

Change 6.11 [basic.align] paragraph 9 as follows:

> If a request for a specific extended alignment in a specific context is not supported by an implementation, the ~~implementation may reject the request as~~ **program is** ill-formed. ~~The implementation may also silently ignore the requested alignment. [Note: a~~**A**dditionally, a request for runtime allocation of dynamic ~~memory~~ **storage** for which the requested alignment cannot be honored ~~may~~ **shall** be treated as an allocation failure. ~~— *end note*~~]

---

519. Null pointer preservation in `void*` conversions

Section: 7.11 [conv.ptr] **Status:** CD1 **Submitter:** comp.std.c++ **Date:** 19 May 2005

[Voted into WP at April, 2006 meeting.]

The C standard says in 6.3.2.3, paragraph 4:

> Conversion of a null pointer to another pointer type yields a null pointer of that type. Any two null pointers shall compare equal.

C++ appears to be incompatible with the first sentence in only two areas:

```
A *a = 0;
void *v = a;
```

C++ (7.11 [conv.ptr] paragraph 2) says nothing about the value of `v`.

```
void *v = 0;
A *b = (A*)v; // aka static_cast<A*>(v)
```

C++ (8.2.9 [expr.static.cast] paragraph 10) says nothing about the value of `b`.

Suggested changes:

1. Add the following sentence to 7.11 [conv.ptr] paragraph 2:

   > The null pointer value is converted to the null pointer value of the destination type.

2. Add the following sentence to 8.2.9 [expr.static.cast] paragraph 10:

   > The null pointer value (7.11 [conv.ptr]) is converted to the null pointer value of the destination type.

Proposed resolution (October, 2005):

1. Add the indicated words to 7.11 [conv.ptr] paragraph 2:

   > An rvalue of type "pointer to *cv* `T`," where `T` is an object type, can be converted to an rvalue of type "pointer to *cv* `void`". The result of converting a "pointer to *cv* `T`" to a "pointer to *cv* `void`" points to the start of the storage location where the object of type `T` resides, as if the object is a most derived object (4.5 [intro.object]) of type `T` (that is, not a base class subobject). **The null pointer value is converted to the null pointer value of the destination type.**

2. Add the indicated words to 8.2.9 [expr.static.cast] paragraph 11:

   > An rvalue of type "pointer to *cv1* `void`" can be converted to an rvalue of type "pointer to *cv2* `T`," where `T` is an object type and *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*. **The null pointer value is converted to the null pointer value of the destination type.** A value of type pointer to object converted to "pointer to *cv* `void`" and back, possibly with different cv-qualification, shall have its original value...

---

654. Conversions to and from `nullptr_t`

Section: 7.11 [conv.ptr] **Status:** CD1 **Submitter:** Jason Merrill **Date:** 7 October 2007

[Voted into the WP at the June, 2008 meeting as paper N2656.]

In the interest of promoting use of `nullptr` instead of the integer literal 0 as the null pointer constant, the proposal accepted by the Committee does not provide for converting a zero-valued integral constant to type `std::nullptr_t`. However, this omission reduces the utility of the feature for use in the library for smart pointers. In particular, the addition of that conversion (along with a converting constructor accepting a `std::nullptr_t`) would allow smart pointers to be used just like ordinary pointers in expressions like:

```
if (p == 0) { }
if (0 == p) { }
if (p != 0) { }
if (0 != p) { }
p = 0;
```

The existing use of the "unspecified bool type" idiom supports this usage, but being able to use `std::nullptr_t` instead would be simpler and more elegant.

Jason Merrill: I have another reason to support the conversion as well: it seems to me very odd for `nullptr_t` to be more restrictive than `void*`. Anything we can do with an arbitrary pointer, we ought to be able to do with `nullptr_t` as well. Specifically, since there is a standard conversion from literal 0 to `void*`, and there is a standard conversion from `void*` to `bool`, `nullptr_t` should support the same conversions.

This changes two of the example lines in the proposal as adopted:

```
if (nullptr) ;      // error, no conversion to bool
if (nullptr == 0) ; // error
```

become

```
if (nullptr) ;      // evaluates to false
if( nullptr == 0 ); // evaluates to true
```

And later,

```
char* ch3 = expr ? nullptr : nullptr; // ch3 is the null pointer value
char* ch4 = expr ? 0 : nullptr;       // ch4 is the null pointer value
int n3 = expr ? nullptr : nullptr;    // error, nullptr_t can't be converted to int
int n4 = expr ? 0 : nullptr;          // error, nullptr_t can't be converted to int
```

I would also allow `reinterpret_cast` from `nullptr_t` to integral type, with the same semantics as a `reinterpret_cast` from the null pointer value to integral type.

Basically, I would like `nullptr_t` to act like a `void*` which is constrained to always be `(void*)0`.

---

480. Is a base of a virtual base also virtual?

Section: 7.12 [conv.mem] **Status:** CD1 **Submitter:** Mark Mitchell **Date:** 18 Oct 2004

[Voted into WP at the October, 2006 meeting.]

When the Standard refers to a virtual base class, it should be understood to include base classes of virtual bases. However, the Standard doesn't actually say this anywhere, so when 7.12 [conv.mem] (for example) forbids casting to a derived class member pointer from a virtual base class member pointer, it could be read as meaning:

```
struct B {};
struct D : public B {};
struct D2 : virtual public D {};

int B::*p;
int D::*q;

void f() {
  static_cast<int D2::*>(p);  // permitted
  static_cast<int D2::*>(q);  // forbidden
}
```

Proposed resolution (October, 2005):

1. Change 7.12 [conv.mem] paragraph 2 as indicated:

   > ...If `B` is an inaccessible (clause 14 [class.access]), ambiguous (13.2 [class.member.lookup]) or virtual (13.1 [class.mi]) base class of `D`, **or a base class of a virtual base class of** `D`**,** a program that necessitates this conversion is ill-formed...

2. Change 8.2.9 [expr.static.cast] paragraph 2 as indicated:

   > ...and `B` is ~~not~~ **neither** a virtual base class of `D` **nor a base class of a virtual base class of** `D`...

3. Change 8.2.9 [expr.static.cast] paragraph 9 as indicated:

   > ...and `B` is ~~not~~ **neither** a virtual base class of `D` **nor a base class of a virtual base class of** `D`...

---

222. Sequence points and lvalue-returning operators

Section: 8 [expr] **Status:** CD1 **Submitter:** Andrew Koenig **Date:** 20 Dec 1999

[Voted into the WP at the September, 2008 meeting.]

I believe that the committee has neglected to take into account one of the differences between C and C++ when defining sequence points. As an example, consider

```
(a += b) += c;
```

where `a`, `b`, and `c` all have type `int`. I believe that this expression has undefined behavior, even though it is well-formed. It is not well-formed in C, because += returns an rvalue there. The reason for the undefined behavior is that it modifies the value of `a' twice between sequence points.

Expressions such as this one are sometimes genuinely useful. Of course, we could write this particular example as

```
a += b; a += c;
```

but what about

```
void scale(double* p, int n, double x, double y) {
    for (int i = 0; i < n; ++i) {
        (p[i] *= x) += y;
    }
}
```

All of the potential rewrites involve multiply-evaluating `p[i]` or unobvious circumlocations like creating references to the array element.

One way to deal with this issue would be to include built-in operators in the rule that puts a sequence point between evaluating a function's arguments and evaluating the function itself. However, that might be overkill: I see no reason to require that in

```
x[i++] = y;
```

the contents of `i' must be incremented before the assignment.

A less stringent alternative might be to say that when a built-in operator yields an lvalue, the implementation shall not subsequently change the value of that object as a consequence of that operator.

I find it hard to imagine an implementation that does not do this already. Am I wrong? Is there any implementation out there that does not `do the right thing' already for `(a += b) += c`?

8.18 [expr.ass] paragraph 1 says,

> The result of the assignment operation is the value stored in the left operand after the assignment has taken place; the result is an lvalue.

What is the normative effect of the words "after the assignment has taken place"? I think that phrase ought to mean that in addition to whatever constraints the rules about sequence points might impose on the implementation, assignment operators on built-in types have the additional constraint that they must store the left-hand side's new value before returning a reference to that object as their result.

One could argue that as the C++ standard currently stands, the effect of `x = y = 0;` is undefined. The reason is that it both fetches and stores the value of `y`, and does not fetch the value of `y` in order to compute its new value.

I'm suggesting that the phrase "after the assignment has taken place" should be read as constraining the implementation to set `y` to 0 before yielding the value of `y` as the result of the subexpression `y = 0`.

Note that this suggestion is different from asking that there be a sequence point after evaluation of an assignment. In particular, I am not suggesting that an order constraint be imposed on any side effects other than the assignment itself.

Francis Glassborow:

My understanding is that for a single variable:

1. Multiple read accesses without a write are OK
2. A single read access followed by a single write (of a value dependant on the read, so that the read MUST happen first) is OK
3. A write followed by an actual read is undefined behaviour
4. Multiple writes have undefined behaviour

It is the 3) that is often ignored because in practice the compiler hardly ever codes for the read because it already has that value but in complicated evaluations with a shortage of registers, that is not always the case. Without getting too close to the hardware, I think we both know that a read too close to a write can be problematical on some hardware.

So, in `x = y = 0;`, the implementation must NOT fetch a value from `y`, instead it has to "know" what that value will be (easy because it has just computed that in order to know what it must, at some time, store in `y`). From this I deduce that computing the lvalue (to know where to store) and the rvalue to know what is stored are two entirely independent actions that can occur in any order commensurate with the overall requirements that both operands for an operator be evaluated before the operator is.

Erwin Unruh:

C distinguishes between the resulting value of an assignment and putting the value in store. So in C a compiler might implement the statement `x=y=0;` either as `x=0;y=0;` or as `y=0;x=0;` In C the statement `(x += 5) += 7;` is not allowed because the first `+=` yields an rvalue which is not allowed as left operand to `+=`. So in C an assignment is not a sequence of write/read because the result is not really "read".

In C++ we decided to make the result of assignment an lvalue. In this case we do not have the option to specify the "value" of the result. That is just the variable itself (or its address in a different view). So in C++, strictly speaking, the statement `x=y=0;` must be implemented as `y=0;x=y;` which makes a big difference if `y` is declared `volatile`.

Furthermore, I think undefined behaviour should not be the result of a single mentioning of a variable within an expression. So the statement `(x +=5) += 7;` should NOT have undefined behaviour.

In my view the semantics could be:

1. if the result of an assignment is used as an rvalue, its value is that of the variable after assignment. The actual store takes place before the next sequence point, but may be before the value is used. This is consistent with C usage.
2. if the result of an assignment is used as an lvalue to store another value, then the new value will be stored in the variable before the next sequence point. It is unspecified whether the first assigned value is stored intermediately.
3. if the result of an assignment is used as an lvalue to take an address, that address is given (it doesn't change). The actual store of the new value takes place before the next sequence point.

Jerry Schwarz:

My recollection is different from Erwin's. I am confident that the intention when we decided to make assignments lvalues was not to change the semantics of evaluation of assignments. The semantics was supposed to remain the same as C's.

Ervin seems to assume that because assignments are lvalues, an assignment's value must be determined by a read of the location. But that was definitely not our intention. As he notes this has a significant impact on the semantics of assignment to a volatile variable. If Erwin's interpretation were correct we would have no way to write a volatile variable without also reading it.

Lawrence Crowl:

For `x=y=0`, lvalue semantics implies an lvalue to rvalue conversion on the result of `y=0`, which in turn implies a read. If `y` is `volatile`, lvalue semantics implies both a read and a write on `y`.

The standard apparently doesn't state whether there is a value dependence of the lvalue result on the completion of the assignment. Such a statement in the standard would solve the non-volatile C compatibility issue, and would be consistent with a user-implemented `operator=`.

Another possible approach is to state that primitive assignment operators have two results, an lvalue and a corresponding "after-store" rvalue. The rvalue result would be used when an rvalue is required, while the lvalue result would be used when an lvalue is required. However, this semantics is unsupportable for user-defined assignment operators, or at least inconsistent with all implementations that I know of. I would not enjoy trying to write such two-faced semantics.

Erwin Unruh:

The intent was for assignments to behave the same as in C. Unfortunately the change of the result to lvalue did not keep that. An "lvalue of type int" has no "int" value! So there is a difference between intent and the standard's wording.

So we have one of several choices:

- live with the incompatibility (and the problems it has for volatile variables)
- make the result of assignment an rvalue (only builtin-assignment, maybe only for builtin types), which makes some presently valid programs invalid
- introduce "two-face semantics" for builtin assignments, and clarify the sequence problematics
- make a special rule for assignment to a volatile lvalue of builtin type

I think the last one has the least impact on existing programs, but it is an ugly solution.

Andrew Koenig:

Whatever we may have intended, I do not think that there is any clean way of making

```
volatile int v;
int i;

i = v = 42;
```

have the same semantics in C++ as it does in C. Like it or not, the subexpression `v = 42` has the type ``reference to volatile int,'' so if this statement has any meaning at all, the meaning must be to store 42 in `v` and then fetch the value of `v` to assign it to i.

Indeed, if `v` is volatile, I cannot imagine a conscientious programmer writing a statement such as this one. Instead, I would expect to see

```
v = 42;
i = v;
```

if the intent is to store 42 in `v` and then fetch the (possibly changed) value of `v`, or

```
v = 42;
i = 42;
```

if the intent is to store 42 in both `v` and `i`.

What I do want is to ensure that expressions such as ``i = v = 42'' have well-defined semantics, as well as expressions such as `(i = v) = 42` or, more realistically, `(i += v) += 42`.

I wonder if the following resolution is sufficient:

Append to 8.18 [expr.ass] paragraph 1:

> There is a sequence point between assigning the new value to the left operand and yielding the result of the assignment expression.

I believe that this proposal achieves my desired effect of not constraining when `j` is incremented in `x[j++] = y`, because I don't think there is a constraint on the relative order of incrementing `j` and executing the assignment. However, I do think it allows expressions such as `(i += v) += 42`, although with different semantics from C if `v` is volatile.

Notes on 10/01 meeting:

There was agreement that adding a sequence point is probably the right solution.

Notes from the 4/02 meeting:

The working group reaffirmed the sequence-point solution, but we will look for any counter-examples where efficiency would be harmed.

For drafting, we note that `++x` is defined in 8.3.2 [expr.pre.incr] as equivalent to `x+=1` and is therefore affected by this change. `x++` is not affected. Also, we should update any list of all sequence points.

Notes from October 2004 meeting:

Discussion centered around whether a sequence point "between assigning the new value to the left operand and yielding the result of the expression" would require completion of all side effects of the operand expressions before the value of the assignment expression was used in another expression. The consensus opinion was that it would, that this is the definition of a sequence point. Jason Merrill pointed out that adding a sequence point after the assignment is essentially the same as rewriting

```
b += a
```

as

```
b += a, b
```

Clark Nelson expressed a desire for something like a "weak" sequence point that would force the assignment to occur but that would leave the side effects of the operands unconstrained. In support of this position, he cited the following expression:

```
j = (i = j++)
```

With the proposed addition of a full sequence point after the assignment to `i`, the net effect is no change to `j`. However, both g++ and MSVC++ behave differently: if the previous value of `j` is 5, the value of the expression is 5 but `j` gets the value 6.

Clark Nelson will investigate alternative approaches and report back to the working group.

Proposed resolution (March, 2008):

This issue is resolved by the adoption of the sequencing rules and the resolution of issue 637.

---

351. Sequence point error: unspecified or undefined?

Section: 8 [expr] **Status:** CD1 **Submitter:** Andrew Koenig **Date:** 23 April 2002

[Voted into WP at March 2004 meeting.]

I have found what looks like a bug in clause 8 [expr], paragraph 4:

> Between the previous and next sequence point a scalar object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be accessed only to determine the value to be stored. The requirements of this paragraph shall be met for each allowable ordering of the subexpressions of a full expression; otherwise the behavior is undefined. Example:
>
> ```
> i = v[i++];                     // the behavior is unspecified
> i = 7, i++, i++;                // i becomes 9
>
> i = ++i + 1;                    // the behavior is unspecified
> i = i + 1;                      // the value of i is incremented
> ```
>
> --end example]

So which is it, unspecified or undefined?

Notes from October 2002 meeting:

We should find out what C99 says and do the same thing.

Proposed resolution (April 2003):

Change the example in clause 8 [expr], paragraph 4 from

> [*Example:*
>
> ```
> i = v[i++];                     //  the behavior is unspecified
> i = 7, i++, i++;                //   i  becomes  9
>
> i = ++i + 1;                    //  the behavior is unspecified
> i = i + 1;                      //  the value of  i  is incremented
> ```
>
> --- end example]

to (changing "unspecified" to "undefined" twice)

> [*Example:*
>
> ```
> i = v[i++];                     //  the behavior is undefined
> i = 7, i++, i++;                //   i  becomes  9
>
> i = ++i + 1;                    //  the behavior is undefined
> i = i + 1;                      //  the value of  i  is incremented
> ```
>
> --- end example]

---

451. Expressions with invalid results and ill-formedness

Section: 8 [expr] **Status:** CD1 **Submitter:** Gennaro Prota **Date:** 19 Jan 2004

[Voted into WP at October 2005 meeting.]

Clause 8 [expr] par. 5 of the standard says:

> If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined, unless such an expression is a constant expression (5.19), in which case the program is ill-formed.

Well, we do know that except in some contexts (e.g. controlling expression of a #if, array bounds), a compiler is not required to evaluate constant-expressions in compile time, right?

Now, let us consider, the following simple snippet:

```
if (a && 1/0)
    ...
```

with a, to fix our attention, being *not* a constant expression. The quote above seems to say that since 1/0 is a constant (sub-)expression, the program is ill-formed. So, is it the intent that such ill-formedness is diagnosable at run-time? Or is it the intent that the above gives undefined behavior (if 1/0 is evaluated) and is not ill-formed?

I think the intent is actually the latter, so I propose the following rewording of the quoted section:

> If an expression is evaluated but its result is not mathematically defined or not in the range of representable values for its type the behavior is undefined, unless such an expression is a constant expression (5.19) **that shall be evaluated during program translation**, in which case the program is ill-formed.

Rationale (March, 2004):

We feel the standard is clear enough. The quoted sentence does begin "If during the evaluation of an expression, ..." so the rest of the sentence does not apply to an expression that is not evaluated.

Note (September, 2004):

Gennaro Prota feels that the CWG missed the point of his original comment: unless a constant expression appears in a context that *requires* a constant expression, an implementation is permitted to defer its evaluation to runtime. An evaluation that fails at runtime cannot affect the well-formedness of the program; only expressions that are evaluated at compile time can make a program ill-formed.

The status has been reset to "open" to allow further discussion.

Proposed resolution (October, 2004):

Change paragraph 5 of 8 [expr] as indicated:

> If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined, unless such an expression ~~is a constant expression~~ **appears where an integral constant expression is required** (8.20 [expr.const]), in which case the program is ill-formed.

---

113. Visibility of called function

Section: 8.2.2 [expr.call] **Status:** CD1 **Submitter:** Christophe de Dinechin **Date:** 5 May 1999

[Moved to DR at 10/01 meeting.]

Christophe de Dinechin: In 8.2.2 [expr.call] , paragraph 2 reads:

> If no declaration of the called function is visible from the scope of the call the program is ill-formed.

I think nothing there or in the previous paragraph indicates that this does not apply to calls through pointer or virtual calls.

Mike Miller: "The called function" is unfortunate phraseology; it makes it sound as if it's referring to the function actually called, as opposed to the identifier in the postfix expression. It's wrong with respect to Koenig lookup, too (the declaration need not be visible if it can be found in a class or namespace associated with one or more of the arguments).

In fact, this paragraph should be a note. There's a general rule that says you have to find an unambiguous declaration of any name that is used (6.4 [basic.lookup] paragraph 1); the only reason this paragraph is here is to contrast with C's implicit declaration of called functions.

Proposed resolution:

Change section 8.2.2 [expr.call] paragraph 2 from:

> If no declaration of the called function is visible from the scope of the call the program is ill-formed.

to:

[*Note:* if a function or member function name is used, and name lookup (6.4 [basic.lookup]) does not find a declaration of that name, the program is ill-formed. No function is implicitly declared by such a call.]

(See also issue 218.)

---

118. Calls via pointers to virtual member functions

Section: 8.2.2 [expr.call] **Status:** CD1 **Submitter:** Martin O'Riordan **Date:** 17 May 1999

[Voted into the WP at the June, 2008 meeting.]

Martin O'Riordan: Having gone through all the relevant references in the IS, it is not conclusive that a call via a pointer to a virtual member function is polymorphic at all, and could legitimately be interpreted as being static.

Consider 8.2.2 [expr.call] paragraph 1:

> The function called in a member function call is normally selected according to the static type of the object expression (clause 13 [class.derived]), but if that function is `virtual` and is not specified using a *qualified-id* then the function actually called will be the final overrider (13.3 [class.virtual]) of the selected function in the dynamic type of the object expression.

Here it is quite specific that you get the polymorphic call only if you use the unqualified syntax. But, the address of a member function is "always" taken using the qualified syntax, which by inference would indicate that call with a PMF is static and not polymorphic! Not what was intended.

Yet other references such as 8.5 [expr.mptr.oper] paragraph 4:

> If the dynamic type of the object does not contain the member to which the pointer refers, the behavior is undefined.

indicate that the opposite may have been intended, by stating that it is the dynamic type and not the static type that matters. Also, 8.5 [expr.mptr.oper] paragraph 6:

> If the result of `.*` or `->*` is a function, then that result can be used only as the operand for the function call operator (). [*Example:*
>
> ```
> (ptr_to_obj->*ptr_to_mfct)(10);
> ```
>
> calls the member function denoted by `ptr_to_mfct` for the object pointed to by `ptr_to_obj`.]

which also implies that it is the object pointed to that determines both the validity of the expression (the static type of '`ptr_to_obj`' may not have a compatible function) and the implicit (polymorphic) meaning. Note too, that this is stated in the non-normative example text.

Andy Sawyer: Assuming the resolution is what I've assumed it is for the last umpteen years (i.e. it does the polymorphic thing), then the follow on to that is "Should there also be a way of selecting the non-polymorphic behaviour"?

Mike Miller: It might be argued that the current wording of 8.2.2 [expr.call] paragraph 1 does give polymorphic behavior to simple calls via pointers to members. (There is no *qualified-id* in `obj.*pmf`, and the IS says that if the function is not specified using a *qualified-id*, the final overrider will be called.) However, it clearly says the wrong thing when the pointer-to-member itself is specified using a *qualified-id* (`obj.*X::pmf`).

Bill Gibbons: The phrase *qualified-id* in 8.2.2 [expr.call] paragraph 1 refers to the *id-expression* and not to the "pointer-to-member expression" earlier in the paragraph:

> For a member function call, the postfix expression shall be an implicit (12.2.2 [class.mfct.non-static] , 12.2.3 [class.static]) or explicit class member access (8.2.5 [expr.ref]) whose *id-expression* is a function member name, or a pointer-to-member expression (8.5 [expr.mptr.oper]) selecting a function member.

Mike Miller: To be clear, here's an example:

```
struct S {
    virtual void f();
};
void (S::*pmf)();
void g(S* sp) {
    sp->f();         // 1: polymorphic
    sp->S::f();      // 2: non-polymorphic
    (sp->S::f)();    // 3: non-polymorphic
    (sp->*pmf)();    // 4: polymorphic
    (sp->*&S::f)();  // 5: polymorphic
}
```

Notes from October 2002 meeting:

This was moved back to open for lack of a champion. Martin O'Riordan is not expected to be attending meetings.

Proposed resolution (February, 2008):

1. Change 8.2.2 [expr.call] paragraph 1 as follows:

> ... For a member function call, the postfix expression shall be an implicit (12.2.2 [class.mfct.non-static], 12.2.3 [class.static]) or explicit class member access (8.2.5 [expr.ref]) whose *id-expression* is a function member name, or a pointer-to-member expression (8.5 [expr.mptr.oper]) selecting a function member~~. The first expression in the postfix expression is then called~~

~~the *object expression,* and~~**;** the call is as a member of the object pointed to or referred to **by the object expression (8.2.5 [expr.ref], 8.5 [expr.mptr.oper])**. In the case of an implicit class member access, the implied object is the one pointed to by `this`. [*Note:* a member function call of the form `f()` is interpreted as `(*this).f()` (see 12.2.2 [class.mfct.non-static]). —*end note*] If a function or member function name is used, the name can be overloaded (clause 16 [over]), in which case the appropriate function shall be selected according to the rules in 16.3 [over.match]. ~~The function called in a member function call is normally selected according to the static type of the object expression (clause 13 [class.derived]), but if that function is `virtual` and is not specified using a *qualified-id* then the function actually called will be the final overrider (13.3 [class.virtual]) of the selected function in the dynamic type of the object expression~~ **If the selected function is non-virtual, or if the *id-expression* in the class member access expression is a *qualified-id*, that function is called. Otherwise, its final overrider (13.3 [class.virtual]) in the dynamic type of the object expression is called.** ...

2. Change 8.5 [expr.mptr.oper] paragraph 4 as follows:

> **The first operand is called the *object expression*.** If the dynamic type of the object **expression** does not contain the member to which the pointer refers, the behavior is undefined.

---

506. Conditionally-supported behavior for non-POD objects passed to ellipsis

Section: 8.2.2 [expr.call] **Status:** CD1 **Submitter:** Mike Miller **Date:** 14 Apr 2005

[Voted into WP at the October, 2006 meeting.]

The current wording of 8.2.2 [expr.call] paragraph 7 states:

> When there is no parameter for a given argument, the argument is passed in such a way that the receiving function can obtain the value of the argument by invoking `va_arg` (21.10 [support.runtime]). The lvalue-to-rvalue (7.1 [conv.lval]), array-to-pointer (7.2 [conv.array]), and function-to-pointer (7.3 [conv.func]) standard conversions are performed on the argument expression. After these conversions, if the argument does not have arithmetic, enumeration, pointer, pointer to member, or class type, the program is ill-formed. If the argument has a non-POD class type (clause 12 [class]), the behavior is undefined.

Paper J16/04-0167=WG21 N1727 suggests that passing a non-POD object to ellipsis be ill-formed. In discussions at the Lillehammer meeting, however, the CWG felt that the newly-approved category of conditionally-supported behavior would be more appropriate.

Proposed resolution (October, 2005):

Change 8.2.2 [expr.call] paragraph 7 as indicated:

> ...After these conversions, if the argument does not have arithmetic, enumeration, pointer, pointer to member, or class type, the program is ill-formed. ~~If the argument has a non-POD class type (clause 9), the behavior is undefined.~~ **Passing an argument of non-POD class type (clause 9) with no corresponding parameter is conditionally-supported, with implementation-defined semantics.**

---

634. Conditionally-supported behavior for non-POD objects passed to ellipsis redux

Section: 8.2.2 [expr.call] **Status:** CD1 **Submitter:** Howard Hinnant **Date:** 6 Jun 2007

[Voted into the WP at the September, 2008 meeting.]

Issue 506 changed passing a non-POD class type to an ellipsis from undefined behavior to conditionally-supported behavior. As a result, an implementation could conceivably reject code like the following:

```
struct two {char _[2];};

template <class From, class To>
struct is_convertible
{
private:
        static From f;

        template <class U> static char test(const U&);
        template <class U> static two test(...);
public:
        static const bool value = sizeof(test<To>(f)) == 1;
};

struct A
{
     A();
};

int main()
{
     const bool b = is_convertible<A,int>::value;  // b == false
}
```

This technique has become popular in template metaprogramming, and no non-POD object is actually passed at runtime. Concepts will eliminate much (perhaps not all) of the need for this kind of programming, but legacy code will persist.

Perhaps this technique should be officially supported by allowing implementations to reject passing a non-POD type to ellipsis only if it appears in a potentially-evaluated expression?

Notes from the July, 2007 meeting:

The CWG agreed with the suggestion to allow such calls in unevaluated contexts.

Proposed resolution (September, 2007):

Change 8.2.2 [expr.call] paragraph 7 as follows:

> ...Passing ~~an~~ **a potentially-evaluated** argument of non-trivial class type (clause 12 [class]) with no corresponding parameter is conditionally-supported, with implementation-defined semantics...

---

466. cv-qualifiers on pseudo-destructor type

Section: 8.2.4 [expr.pseudo] **Status:** CD1 **Submitter:** Mark Mitchell **Date:** 18 Mar 2004

[Voted into WP at April, 2006 meeting.]

8.2.4 [expr.pseudo] paragraph 2 says both:

> The type designated by the pseudo-destructor-name shall be the same as the object type.

and also:

> The cv-unqualified versions of the object type and of the type designated by the pseudo-destructor-name shall be the same type.

Which is it? "The same" or "the same up to cv-qualifiers"? The second sentence is more generous than the first. Most compilers seem to implement the less restrictive form, so I guess that's what I think we should do.

See also issues 305 and 399.

Proposed resolution (October, 2005):

Change 8.2.4 [expr.pseudo] paragraph 2 as follows:

> 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.~~ **The cv-unqualified versions of the object type and of the type designated by the *pseudo-destructor-name* shall be the same type.** Furthermore, the two *type-name*s in a *pseudo-destructor-name* of the form
>
> > $::_{opt}$ *nested-name-specifier*$_{opt}$ *type-name* `::~` *type-name*
>
> shall designate the same scalar type. ~~The cv-unqualified versions of the object type and of the type designated by the *pseudo-destructor-name* shall be the same type.~~

---

421. Is rvalue.field an rvalue?

Section: 8.2.5 [expr.ref] **Status:** CD1 **Submitter:** Gabriel Dos Reis **Date:** 15 June 2003

[Voted into WP at March 2004 meeting.]

Consider

```
typedef
  struct {
    int a;
  } A;

A f(void)
{
  A a;
  return a;
}

int main(void)
{
  int* p = &f().a;   // #1
}
```

Should #1 be rejected? The standard is currently silent.

Mike Miller: I don't believe the Standard is silent on this. I will agree that the wording of 8.2.5 [expr.ref] paragraph 4 bullet 2 is unfortunate, as it is subject to misinterpretation. It reads,

If E1 is an lvalue, then E1.E2 is an lvalue.

The intent is, "and not otherwise."

Notes from October 2003 meeting:

We agree the reference should be an rvalue, and a change along the lines of that recommended by Mike Miller is reasonable.

Proposed Resolution (October 2003):

Change the second bullet of 8.2.5 [expr.ref] paragraph 4 to read:

If E1 is an lvalue, then E1.E2 is an lvalue**; otherwise, it is an rvalue**.

---

492. `typeid` constness inconsistent with example

Section: 8.2.8 [expr.typeid] **Status:** CD1 **Submitter:** Ron Natalie **Date:** 15 Dec 2004

[Voted into WP at April, 2006 meeting.]

There is an inconsistency between the normative text in section 8.2.8 [expr.typeid] and the example that follows.

Here is the relevant passage (starting with paragraph 4):

When `typeid` is applied to a *type-id*, the result refers to a `std::type_info` object representing the type of the *type-id*. If the type of the *type-id* is a reference type, the result of the `typeid` expression refers to a `std::type_info` object representing the referenced type.

The top-level cv-qualifiers of the lvalue expression or the *type-id* that is the operand of `typeid` are always ignored.

and the example:

```
typeid(D) == typeid(const D&); // yields true
```

The second paragraph above says the "*type-id* that is the operand". This would be `const D&`. In this case, the `const` is not at the top-level (i.e., applied to the operand itself).

By a strict reading, the above should yield `false`.

My proposal is that the strict reading of the normative test is correct. The example is wrong. Different compilers here give different answers.

Proposed resolution (April, 2005):

Change the second sentence of 8.2.8 [expr.typeid] paragraph 4 as follows:

If the type of the *type-id* is a reference **to a possibly cv-qualified** type, the result of the `typeid` expression refers to a `std::type_info` object representing the **cv-unqualified** referenced type.

---

54. Static_cast from private base to derived class

Section: 8.2.9 [expr.static.cast] **Status:** CD1 **Submitter:** Steve Adamczyk **Date:** 13 Oct 1998

[Voted into WP at October 2004 meeting.]

Is it okay to use a `static_cast` to cast from a private base class to a derived class? That depends on what the words "valid standard conversion" in paragraph 8 mean — do they mean the conversion exists, or that it would not get an error if it were done? I think the former was intended — and therefore a `static_cast` from a private base to a derived class would be allowed.

Rationale (04/99): A `static_cast` from a private base to a derived class is not allowed outside a member from the derived class, because 7.11 [conv.ptr] paragraph 3 implies that the conversion is not valid. (Classic style casts work.)

Reopened September 2003:

Steve Adamczyk: It makes some sense to disallow the inverse conversion that is pointer-to-member of derived to pointer-to-member of private base. There's less justification for the pointer-to-private-base to pointer-to-derived case. EDG, g++ 3.2, and MSVC++ 7.1 allow the pointer case and disallow the pointer-to-member case. Sun disallows the pointer case as well.

```
struct B {};
struct D : private B {};
int main() {
  B *p = 0;
  static_cast<D *>(p);  // Pointer case: should be allowed
  int D::*pm = 0;
  static_cast<int B::*>(pm);  // Pointer-to-member case: should get error
}
```

There's a tricky case with old-style casts: because the static_cast interpretation is tried first, you want a case like the above to be considered a static_cast, but then issue an error, not be rejected as not a static cast; if you did the latter, you would then try the cast as a reinterpret_cast.

Ambiguity and casting to a virtual base should likewise be errors after the static_cast interpretation is selected.

Notes from the October 2003 meeting:

There was lots of sentiment for making things symmetrical: the pointer case should be the same as the pointer-to-member case. g++ 3.3 now issues errors on both cases.

We decided an error should be issued on both cases. The access part of the check should be done later; by some definition of the word the static_cast is valid, and then later an access error is issued. This is similar to the way standard conversions work.

Proposed Resolution (October 2003):

Replace paragraph 8.2.9 [expr.static.cast]/6:

> The inverse of any standard conversion sequence (clause 7 [conv]), other than the lvalue-to-rvalue (7.1 [conv.lval]), array-to-pointer (7.2 [conv.array]), function-to-pointer (7.3 [conv.func]), and boolean (7.13 [conv.fctptr]) conversions, can be performed explicitly using `static_cast`. The lvalue-to-rvalue (7.1 [conv.lval]), array-to-pointer (7.2 [conv.array]), and function-to-pointer (7.3 [conv.func]) conversions are applied to the operand. Such a `static_cast` is subject to the restriction that the explicit conversion does not cast away constness (8.2.11 [expr.const.cast]), and the following additional rules for specific cases:

with two paragraphs:

> The inverse of any standard conversion sequence (clause 7 [conv]), other than the lvalue-to-rvalue (7.1 [conv.lval]), array-to-pointer (7.2 [conv.array]), function-to-pointer (7.3 [conv.func]), and boolean (7.13 [conv.fctptr]) conversions, can be performed explicitly using `static_cast`. **A program is ill-formed if it uses `static_cast` to perform the inverse of an ill-formed standard conversion sequence.[*Example:***
>
> ```
> struct B {};
> struct D : private B {};
> void f() {
>   static_cast<D*>((B*)0); // Error: B is a private base of D.
>   static_cast<int B::*>((int D::*)0); // Error: B is a private base of D.
> }
> ```
>
> **--- end example]**
>
> The lvalue-to-rvalue (7.1 [conv.lval]), array-to-pointer (7.2 [conv.array]), and function-to-pointer (7.3 [conv.func]) conversions are applied to the operand. Such a `static_cast` is subject to the restriction that the explicit conversion does not cast away constness (8.2.11 [expr.const.cast]), and the following additional rules for specific cases:

In addition, modify the second sentence of 8.4 [expr.cast]/5. The first two sentences of 8.4 [expr.cast]/5 presently read:

> The conversions performed by
>
> - a `const_cast` (5.2.11),
> - a `static_cast` (5.2.9),
> - a `static_cast` followed by a `const_cast`,
> - a `reinterpret_cast` (5.2.10), or
> - a `reinterpret_cast` followed by a `const_cast`,
>
> can be performed using the cast notation of explicit type conversion. The same semantic restrictions and behaviors apply.

Change the second sentence to read:

> The same semantic restrictions and behaviors apply**, with the exception that in performing a `static_cast` in the following situations the conversion is valid even if the base class is inaccessible:**
>
> - **a pointer to an object of derived class type or an lvalue of derived class type may be explicitly converted to a pointer or reference to an unambiguous base class type, respectively;**
> - **a pointer to member of derived class type may be explicitly converted to a pointer to member of an unambiguous non-virtual base class type;**
> - **a pointer to an object of an unambiguous non-virtual base class type, an lvalue of an unambiguous non-virtual base class type, or a pointer to member of an unambiguous non-virtual base class type may be explicitly converted to a pointer, a reference, or a pointer to member of a derived class type, respectively.**

Remove paragraph 8.4 [expr.cast]/7, which presently reads:

> In addition to those conversions, the following `static_cast` and `reinterpret_cast` operations (optionally followed by a `const_cast` operation) may be performed using the cast notation of explicit type conversion, even if the base class type is not accessible:
>
> - a pointer to an object of derived class type or an lvalue of derived class type may be explicitly converted to a pointer or reference to an unambiguous base class type, respectively;
> - a pointer to member of derived class type may be explicitly converted to a pointer to member of an unambiguous non-virtual base class type;
> - a pointer to an object of non-virtual base class type, an lvalue of non-virtual base class type, or a pointer to member of non-virtual base class type may be explicitly converted to a pointer, a reference, or a pointer to member of a derived class

type, respectively.

---

427. `static_cast` **ambiguity: conversion versus cast to derived**

Section: 8.2.9 [expr.static.cast] **Status:** CD1 **Submitter:** Mark Mitchell **Date:** 5 July 2003

[Voted into WP at October 2004 meeting.]

Consider this code:

```
struct B {};

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

extern B& b;

void f() {
  static_cast<const D&>(b);
}
```

The rules for `static_cast` permit the conversion to "const D&" in two ways:

1. D is derived from B, and b is an lvalue, so a cast to D& is OK.
2. const D& t = b is valid, using the constructor for D. [Ed. note: actually, this should be parenthesized initialization.]

The first alternative is 8.2.9 [expr.static.cast]/5; the second is 8.2.9 [expr.static.cast]/2.

Presumably the first alternative is better -- it's the "simpler" conversion. The standard does not seem to make that clear.

Steve Adamczyk: I take the "Otherwise" at the beginning of 8.2.9 [expr.static.cast]/3 as meaning that the paragraph 2 interpretation is used if available, which means in your example above interpretation 2 would be used. However, that's not what EDG's compiler does, and I agree that it's not the "simpler" conversion.

Proposed Resolution (October 2003):

Move paragraph 5.2.9/5:

> An lvalue of type ``*cv1* `B`'', where `B` is a class type, can be cast to type ``reference to *cv2* `D`'', where `D` is a class derived (clause 13 [class.derived]) from `B`, if a valid standard conversion from ``pointer to `D`'' to ``pointer to `B`'' exists (7.11 [conv.ptr]), *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*, and `B` is not a virtual base class of `D`. The result is an lvalue of type ``*cv2* `D`.'' If the lvalue of type ``*cv1* `B`'' is actually a sub-object of an object of type `D`, the lvalue refers to the enclosing object of type `D`. Otherwise, the result of the cast is undefined. [*Example:*
>
> ```
> struct B {};
> struct D : public B {};
> D d;
> B &br = d;
>
> static_cast<D&>(br);          //  produces lvalue to the original  d  object
> ```
>
> --- end example]

before paragraph 8.2.9 [expr.static.cast]/2.

Insert **Otherwise,** before the text of paragraph 8.2.9 [expr.static.cast]/2 (which will become 8.2.9 [expr.static.cast]/3 after the above insertion), so that it reads:

> **Otherwise**, an expression `e` can be explicitly converted to a type `T` using a `static_cast` of the form `static_cast<T>(e)` if the declaration "`T t(e);`" is well-formed, for some invented temporary variable `t` (11.6 [dcl.init]). The effect of such an explicit conversion is the same as performing the declaration and initialization and then using the temporary variable as the result of the conversion. The result is an lvalue if `T` is a reference type (11.3.2 [dcl.ref]), and an rvalue otherwise. The expression `e` is used as an lvalue if and only if the initialization uses it as an lvalue.

---

439. Guarantees on casting pointer back to cv-qualified version of original type

Section: 8.2.9 [expr.static.cast] **Status:** CD1 **Submitter:** Mark Mitchell **Date:** 30 Oct 2003

[Voted into WP at April 2005 meeting.]

Paragraph 8.2.9 [expr.static.cast] paragraph 10 says that:

> A value of type pointer to object converted to "pointer to *cv* `void`" and back to the original pointer type will have its original value.

That guarantee should be stronger. In particular, given:

```
T* p1 = new T;
const T* p2 = static_cast<const T*>(static_cast<void *>(p1));
if (p1 != p2)
  abort ();
```

there should be no call to "abort". The last sentence of Paragraph 8.2.9 [expr.static.cast] paragraph 10 should be changed to read:

> A value of type pointer to object converted to "pointer to *cv* `void`" and back to the original pointer type (or a variant of the original pointer type that differs only in the *cv*-qualifiers applied to the object type) will have its original value. [*Example:*
>
> ```
> T* p1 = new T;
> const T* p2 = static_cast<const T*>(static_cast<void *>(p1));
> bool b = p1 == p2; // b will have the value true.
> ```
>
> *---end example.*]

Proposed resolution:

Change 8.2.9 [expr.static.cast] paragraph 10 as indicated:

> A value of type pointer to object converted to "pointer to *cv* `void`" and back to the original pointer type**, possibly with different cv-qualification,** will have its original value. **[*Example:***
>
> ```
> T* p1 = new T;
> const T* p2 = static_cast<const T*>(static_cast<void *>(p1));
> bool b = p1 == p2; // b will have the value true.
> ```
>
> **---*end example*]**

Rationale: The wording "possibly with different cv-qualification" was chosen over the suggested wording to allow for changes in cv-qualification at different levels in a multi-level pointer, rather than only at the object type level.

---

671. Explicit conversion from a scoped enumeration type to integral type

Section: 8.2.9 [expr.static.cast] **Status:** CD1 **Submitter:** Daveed Vandevoorde **Date:** 22 December 2007

[Voted into the WP at the September, 2008 meeting.]

There appears to be no provision in the Standard for explicit conversion of a value of a scoped enumeration type to an integral type, even though the inverse conversion is permitted. That is,

```
enum class E { e };
static_cast<E>(0);       // #1: OK
static_cast<int>(E::e);  // #2: error
```

This is because values of scope enumeration types (intentionally) cannot be implicitly converted to integral types (7.6 [conv.prom] and 7.8 [conv.integral]) and 8.2.9 [expr.static.cast] was not updated to permit #2, although #1 is covered by paragraph 8.

Proposed resolution (June, 2008):

Add the following as a new paragraph following 8.2.9 [expr.static.cast] paragraph 8:

> A value of a scoped enumeration type (10.2 [dcl.enum]) can be explicitly converted to an integral type. The value is unchanged if the original value can be represented by the specified type. Otherwise, the resulting value is unspecified.

---

195. Converting between function and object pointers

Section: 8.2.10 [expr.reinterpret.cast] **Status:** CD1 **Submitter:** Steve Clamage **Date:** 12 Jan 2000

[Voted into WP at April 2005 meeting.]

It is currently not permitted to cast directly between a pointer to function type and a pointer to object type. This conversion is not listed in 8.2.9 [expr.static.cast] and 8.2.10 [expr.reinterpret.cast] and thus requires a diagnostic to be issued. However, if a sufficiently long integral type exists (as is the case in many implementations), it is permitted to cast between pointer to function types and pointer to object types using that integral type as an intermediary.

In C the cast results in undefined behavior and thus does not require a diagnostic, and Unix C compilers generally do not issue one. This fact is used in the definition of the standard Unix function `dlsym`, which is declared to return `void*` but in fact may return either a pointer to a function or a pointer to an object. The fact that C++ compilers are required to issue a diagnostic is viewed as a "competitive disadvantage" for the language.

Suggested resolution: Add wording to 8.2.10 [expr.reinterpret.cast] allowing conversions between pointer to function and pointer to object types, if the implementation has an integral data type that can be used as an intermediary.

Several points were raised in opposition to this suggestion:

1. Early C++ supported this conversion and it was deliberately removed during the standardization process.
2. The existence of an appropriate integral type is irrelevant to whether the conversion is "safe." The condition should be on whether information is lost in the conversion or not.
3. There are numerous ways to address the problem at an implementation level rather than changing the language. For example, the compiler could recognize the specific case of `dlsym` and omit the diagnostic, or the C++ binding to `dlsym` could be changed (using templates, for instance) to circumvent the violation.
4. The conversion is, in fact, not supported by C; the `dlsym` function is simply relying on non-mandated characteristics of C implementations, and we would be going beyond the requirements of C compatibility in requiring (some) implementations to support the conversion.
5. This issue is in fact not a defect (omitted or self-contradictory requirements) in the current Standard; the proposed change would actually be an extension and should not be considered until the full review of the IS.
6. `dlsym` appears not to be used very widely, and the declaration in the header file is not problematic, only calls to it. Since C code generally requires some porting to be valid C++ anyway, this should be considered one of those items that requires porting.

Martin O'Riordan suggested an alternative approach:

- Do not allow the conversions to be performed via old-style casts.
- Allow `reinterpret_cast` to perform the conversions with the C-style semantics (undefined behavior if not supported).
- Allow `dynamic_cast` to perform the conversions with well-defined behavior: the result would be a null pointer if the implementation could not support the requested conversion.

The advantage of this approach is that it would permit writing portable, well-defined programs involving such conversions. However, it breaks the current degree of compatibility between old and new casts, and it adds functionality to `dynamic_cast` which is not obviously related to its current meaning.

Notes from 04/00 meeting:

Andrew Koenig suggested yet another approach: specify that "no diagnostic is required" if the implementation supports the conversion.

Later note:

It was observed that conversion between function and data pointers is listed as a "common extension" in C99.

Notes on the 10/01 meeting:

It was decided that we want the conversion defined in such a way that it always exists but is always undefined (as opposed to existing only when the size relationship is appropriate, and being implementation-defined in that case). This would allow an implementation to issue an error at compile time if the conversion does not make sense.

Bill Gibbons notes that the definitions of the other similar casts are inconsistent in this regard. Perhaps they should be updated as well.

Proposed resolution (April 2003):

After 8.2.10 [expr.reinterpret.cast] paragraph 6, insert:

> A pointer to a function can be explicitly converted to a pointer to a function of a different type. The effect of calling a function through a pointer to a function type (11.3.5 [dcl.fct]) that is not the same as the type used in the definition of the function is undefined. Except that converting an rvalue of type ``pointer to `T1`'' to the type ``pointer to `T2`'' (where `T1` and `T2` are function types) and back to its original type yields the original pointer value, the result of such a pointer conversion is unspecified. [Note: see also 7.11 [conv.ptr] for more details of pointer conversions.] **It is implementation defined whether a conversion from pointer to object to pointer to function and/or a conversion from pointer to function to pointer to object exist.**

and in paragraph 10:

> An lvalue expression of type `T1` can be cast to the type ``reference to `T2`'' if `T1` **and** `T2` **are object types and** an expression of type ``pointer to `T1`'' can be explicitly converted to the type ``pointer to `T2`'' using a reinterpret_cast. That is, a reference cast `reinterpret_cast< T& >(x)` has the same effect as the conversion `*reinterpret_cast< T* >(&x)` with the built-in `&` and `*` operators. The result is an lvalue that refers to the same object as the source lvalue, but with a different type. No temporary is created, no copy is made, and constructors (15.1 [class.ctor]) or conversion functions (15.3 [class.conv]) are not called.

Drafting Note:

If either conversion exists, the implementation already has to define the behavior (paragraph 3).

Notes from April 2003 meeting:

The new consensus is that if the implementation allows this cast, pointer-to-function converted to pointer-to-object converted back to the original pointer-to-function should work; anything else is undefined behavior. If the implementation does not allow the cast, it should be ill-formed.

Tom Plum is investigating a new concept, that of a "conditionally-defined" feature, which may be applicable here.

Proposed Resolution (October, 2004):

(See paper J16/04-0067 = WG21 N1627 for background material and rationale for this resolution. The resolution proposed here differs only editorially from the one in the paper.)

1. Insert the following into 3 [intro.defs] and renumber all following definitions accordingly:

   1.3.2 conditionally-supported behavior

   behavior evoked by a program construct that is not a mandatory requirement of this International Standard. If a given implementation supports the construct, the behavior shall be as described herein; otherwise, the implementation shall document that the construct is not supported and shall treat a program containing an occurrence of the construct as ill-formed (3 [intro.defs]).

2. Add the indicated words to 4.1 [intro.compliance] paragraph 2, bullet 2:

   - If a program contains a violation of any diagnosable rule, **or an occurrence of a construct described herein as "conditionally-supported" or as resulting in "conditionally-supported behavior" when the implementation does not in fact support that construct**, a conforming implementation shall issue at least one diagnostic message, except that

3. Insert the following as a new paragraph following 8.2.10 [expr.reinterpret.cast] paragraph 7:

   Converting a pointer to a function to a pointer to an object type or vice versa evokes conditionally-supported behavior. In any such conversion supported by an implementation, converting from an rvalue of one type to the other and back (possibly with different cv-qualification) shall yield the original pointer value; mappings between pointers to functions and pointers to objects are otherwise implementation-defined.

4. Change 10.4 [dcl.asm] paragraph 1 as indicated:

   ~~The meaning of an~~ **An** `asm` declaration **evokes conditionally-supported behavior. If supported, its meaning** is implementation-defined.

5. Change 10.5 [dcl.link] paragraph 2 as indicated:

   The *string-literal* indicates the required language linkage. ~~The meaning of the string-literal is implementation-defined. A linkage-specification with a string that is unknown to the implementation is ill-formed.~~ **This International Standard specifies the semantics of C and C++ language linkage. Other values of the *string-literal* evoke conditionally-supported behavior, with implementation-defined semantics. [*Note:* Therefore, a *linkage-specification* with a *string-literal* that is unknown to the implementation requires a diagnostic.** ~~When the string-literal in a linkage-specification names a programming language, the spelling of the programming language's name is implementation-defined. [Note:~~ **It** is recommended that the spelling be taken from the document defining that language, for example `Ada` (not `ADA`) and `Fortran` or `FORTRAN` (depending on the vintage). ~~The semantics of a language linkage other than C++ or C are implementation-defined.~~]

6. Change 17 [temp] paragraph 4 as indicated:

   A template, a template explicit specialization (17.8.3 [temp.expl.spec]), or a class template partial specialization shall not have C linkage. If the linkage of one of these is something other than C or C++, the ~~behavior is implementation-defined~~ **result is conditionally-supported behavior, with implementation-defined semantics**.

---

463. `reinterpret_cast<T*>(0)`

Section: 8.2.10 [expr.reinterpret.cast] **Status:** CD1 **Submitter:** Gennaro Prota **Date:** 14 Feb 2004

[Voted into WP at April, 2006 meeting.]

Is `reinterpret_cast<T*>(`*null_pointer_constant*`)` guaranteed to yield the null pointer value of type T*?

I think a committee clarification is needed. Here's why: 8.2.10 [expr.reinterpret.cast] par. 8 talks of "null pointer value", not "null pointer constant", so it would seem that

```
reinterpret_cast<T*>(0)
```

is a normal int->T* conversion, with an implementation-defined result.

However a little note to 8.2.10 [expr.reinterpret.cast] par. 5 says:

> Converting an integral constant expression (5.19) with value zero always yields a null pointer (4.10), but converting other expressions that happen to have value zero need not yield a null pointer.

Where is this supported in normative text? It seems that either the footnote or paragraph 8 doesn't reflect the intent.

SUGGESTED RESOLUTION: I think it would be better to drop the footnote #64 (and thus the special case for ICEs), for two reasons:

a) it's not normative anyway; so I doubt anyone is relying on the guarantee it hints at, unless that guarantee is given elsewhere in a normative part

b) users expect reinterpret_casts to be almost always implementation dependent, so this special case is a surprise. After all, if one wants a null pointer there's static_cast. And if one wants reinterpret_cast semantics the special case requires doing some explicit cheat, such as using a non-const variable as intermediary:

```
int v = 0;
reinterpret_cast<T*>(v); // implementation defined
```

```
reinterpret_cast<T*>(0); // null pointer value of type T*
const int w = 0;
reinterpret_cast<T*>(w); // null pointer value of type T*
```

It seems that not only that's providing a duplicate functionality, but also at the cost to hide what seems the more natural one.

Notes from October 2004 meeting:

This footnote was added in 1996, after the invention of `reinterpret_cast`, so the presumption must be that it was intentional. At this time, however, the CWG feels that there is no reason to require that `reinterpret_cast<T*>(0)` produce a null pointer value as its result.

Proposed resolution (April, 2005):

1. Delete the footnote in 8.2.10 [expr.reinterpret.cast] paragraph 5 reading,

   > Converting an integral constant expression (8.20 [expr.const]) with value zero always yields a null pointer (7.11 [conv.ptr]), but converting other expressions that happen to have value zero need not yield a null pointer.

2. Add the indicated note to 8.2.10 [expr.reinterpret.cast] paragraph 8:

   > The null pointer value (7.11 [conv.ptr]) is converted to the null pointer value of the destination type. **[*Note:* A null pointer constant, which has integral type, is not necessarily converted to a null pointer value. —*end note*]**

---

324. Can "`&`" be applied to assignment to bit-field?

Section: 8.3.1 [expr.unary.op] **Status:** CD1 **Submitter:** Alasdair Grant **Date:** 27 Nov 2001

[Voted into WP at October 2003 meeting.]

An assignment returns an lvalue for its left operand. If that operand refers to a bit field, can the "`&`" operator be applied to the assignment? Can a reference be bound to it?

```
struct S { int a:3; int b:3; int c:3; };

void f()
{
  struct S s;
  const int *p = &(s.b = 0);     // (a)
  const int &r = (s.b = 0);      // (b)
        int &r2 = (s.b = 0);     // (c)
}
```

Notes from the 4/02 meeting:

The working group agreed that this should be an error.

Proposed resolution (October 2002):

In 8.3.2 [expr.pre.incr] paragraph 1 (prefix "`++`" and "`--`" operators), change

> The value is the new value of the operand; it is an lvalue.

to

> The result is the updated operand; it is an lvalue, and it is a bit-field if the operand is a bit-field.

In 8.16 [expr.cond] paragraph 4 ("`?`" operator), add the indicated text:

> If the second and third operands are lvalues and have the same type, the result is of that type and is an lvalue **and it is a bit-field if the second or the third operand is a bit-field, or if both are bit-fields.**

In 8.18 [expr.ass] paragraph 1 (assignment operators) add the indicated text (the original text is as updated by issue 221, which is DR but not in TC1):

> The assignment operator (`=`) and the compound assignment operators all group right-to-left. All require a modifiable lvalue as their left operand and return an lvalue with the type and value of the left operand after the assignment has taken place. **The result in all cases is a bit-field if the left operand is a bit-field.**

Note that issue 222 adds (non-conflicting) text at the end of this same paragraph (8.18 [expr.ass] paragraph 1).

In 8.19 [expr.comma] paragraph 1 (comma operator), change:

> The type and value of the result are the type and value of the right operand; the result is an lvalue if its right operand is.

to

> The type and value of the result are the type and value of the right operand; the result is an lvalue if the right operand is an lvalue, and is a bit-field if the right operand is an lvalue and a bit-field.

Relevant related text (no changes required):

8.3.1 [expr.unary.op] paragraph 4:

> The operand of `&` shall not be a bit-field.

11.6.3 [dcl.init.ref] paragraph 5, bullet 1, sub-bullet 1 (regarding binding a reference to an lvalue):

> ... is an lvalue (but is not a bit-field) ...

---

256. Overflow in size calculations

Section: 8.3.4 [expr.new] **Status:** CD1 **Submitter:** James Kanze **Date:** 15 Oct 2000

[Voted into the WP at the September, 2008 meeting.]

[Picked up by evolution group at October 2002 meeting.]

(See also issue 476.)

The size requested by an array allocation is computed by multiplying the number of elements requested by the size of each element and adding an implementation-specific amount for overhead. It is possible for this calculation to overflow. Is an implementation required to detect this situation and, for instance, throw `std::bad_alloc`?

On one hand, the maximum allocation size is one of the implementation limits specifically mentioned in Annex B [implimits], and, according to 4.1 [intro.compliance] paragraph 2, an implementation is only required to "accept and correctly execute" programs that do not violate its resource limits.

On the other hand, it is difficult or impossible for user code to detect such overflows in a portable fashion, especially given that the array allocation overhead is not fixed, and it would be a service to the user to handle this situation gracefully.

Rationale (04/01):

Each implementation is required to document the maximum size of an object (Annex B [implimits]). It is not difficult for a program to check array allocations to ensure that they are smaller than this quantity. Implementations can provide a mechanism in which users concerned with this problem can request extra checking before array allocations, just as some implementations provide checking for array index and pointer validity. However, it would not be appropriate to require this overhead for every array allocation in every program.

(See issue 624 for a request to reconsider this resolution.)

Note (March, 2008):

The Evolution Working Group has accepted the intent of this issue and referred it to CWG for action for C++0x (see paper J16/07-0033 = WG21 N2173).

Proposed resolution (September, 2008):

This issue is resolved by the resolution of issue 624, given in paper N2757.

---

299. Conversion on array bound expression in `new`

Section: 8.3.4 [expr.new] **Status:** CD1 **Submitter:** Mark Mitchell **Date:** 19 Jul 2001

[Voted into WP at October 2005 meeting.]

In 8.3.4 [expr.new], the standard says that the expression in an array-new has to have integral type. There's already a DR (issue 74) that says it should also be allowed to have enumeration type. But, it should probably also say that it can have a class type with a single conversion to integral type; in other words the same thing as in 9.4.2 [stmt.switch] paragraph 2.

Suggested resolution:

In 8.3.4 [expr.new] paragraph 6, replace "integral or enumeration type (6.9.1 [basic.fundamental])" with "integral or enumeration type (6.9.1 [basic.fundamental]), or a class type for which a single conversion function to integral or enumeration type exists".

Proposed resolution (October, 2004):

Change 8.3.4 [expr.new] paragraph 6 as follows:

> Every *constant-expression* in a *direct-new-declarator* shall be an integral constant expression (8.20 [expr.const]) and evaluate to a strictly positive value. The *expression* in a *direct-new-declarator* shall ~~have~~ **be of** integral type, ~~or~~ enumeration type~~ (3.9.1)~~**, or a class type for which a single conversion function to integral or enumeration type exists (15.3 [class.conv]). If the expression is of class type, the expression is converted by calling the conversion function, and the result of the conversion is used in place of the original expression. The value of the expression shall be**~~with a~~ non-negative~~ value~~. [*Example:* ...

Proposed resolution (April, 2005):

Change 8.3.4 [expr.new] paragraph 6 as follows:

> Every *constant-expression* in a *direct-new-declarator* shall be an integral constant expression (8.20 [expr.const]) and evaluate to a strictly positive value. The *expression* in a *direct-new-declarator* shall ~~have integral or enumeration type (6.9.1 [basic.fundamental]) with a non-negative value~~ **be of integral type, enumeration type, or a class type for which a single conversion function to integral or enumeration type exists (15.3 [class.conv]). If the expression is of class type, the expression is converted by calling that conversion function, and the result of the conversion is used in place of the original expression. If the value of the expression is negative, the behavior is undefined**. [*Example:* ...

---

429. Matching deallocation function chosen based on syntax or signature?

Section: 8.3.4 [expr.new] **Status:** CD1 **Submitter:** John Wilkinson **Date:** 18 July 2003

[Voted into WP at October 2004 meeting.]

What does this example do?

```
#include <stdio.h>
#include <stdlib.h>

struct A {
      void* operator new(size_t alloc_size, size_t dummy=0) {
              printf("A::operator new()\n");
              return malloc(alloc_size);
      };

      void operator delete(void* p, size_t s) {
              printf("A::delete %d\n", s);
      };


      A()  {printf("A constructing\n"); throw 2;};

};

int
main() {
  try {
      A* ap = new A;
      delete ap;
  }
  catch(int) {printf("caught\n"); return 1;}
}
```

The fundamental issue here is whether the deletion-on-throw is driven by the syntax of the new (placement or non-placement) or by signature matching. If the former, the operator delete would be called with the second argument equal to the size of the class. If the latter, it would be called with the second argument 0.

Core issue 127 (in TC1) dealt with this topic. It removed some wording in 18.2 [except.ctor] paragraph 2 that implied a syntax-based interpretation, leaving wording in 8.3.4 [expr.new] paragraph 19 that is signature-based. But there is no accompanying rationale to confirm an explicit choice of the signature-based approach.

EDG and g++ get 0 for the second argument, matching the presumed core issue 127 resolution. But maybe this should be revisited.

Notes from October 2003 meeting:

There was widespread agreement that the compiler shouldn't just silently call the delete with either of the possible values. In the end, we decided it's smarter to issue an error on this case and force the programmer to say what he means.

Mike Miller's analysis of the status quo: 6.7.4.2 [basic.stc.dynamic.deallocation] paragraph 2 says that "`operator delete(void*, std::size_t)`" is a "usual (non-placement) deallocation function" if the class does not declare "`operator delete(void*)`." 6.7.4.1 [basic.stc.dynamic.allocation] does not use the same terminology for allocation functions, but the most reasonable way to understand the uses of the term "placement allocation function" in the Standard is as an allocation function that has more than one parameter and thus can (but need not) be called using the "new-placement" syntax described in 8.3.4 [expr.new]. (In considering issue 127, the core group discussed and endorsed the position that, "If a placement allocation function has default arguments for all its parameters except the first, it can be called using non-placement syntax.")

8.3.4 [expr.new] paragraph 19 says that any non-placement deallocation function matches a non-placement allocation function, and that a placement deallocation function matches a placement allocation function with the same parameter types after the first -- i.e., a non-placement deallocation function cannot match a placement allocation function. This makes sense, because non-placement ("usual") deallocation functions expect to free memory obtained from the system heap, which might not be the case for storage resulting from calling a placement allocation function.

According to this analysis, the example shows a placement allocation function and a non-placement deallocation function, so the deallocation function should not be invoked at all, and the memory will just leak.

Proposed Resolution (October 2003):

Add the following text at the end of 8.3.4 [expr.new] paragraph 19:

If the lookup finds the two-parameter form of a usual deallocation function (6.7.4.2 [basic.stc.dynamic.deallocation]), and that function, considered as a placement deallocation function, would have been selected as a match for the allocation function, the program is ill-formed. [*Example:*

```
struct S {
  // Placement allocation function:
  static void* operator new(std::size_t, std::size_t);

  // Usual (non-placement) deallocation function:
  static void operator delete(void*, std::size_t);
};

S* p = new (0) S; // ill-formed: non-placement deallocation function matches
                  // placement allocation function
```

--- end example]

---

624. Overflow in calculating size of allocation

Section: 8.3.4 [expr.new] **Status:** CD1 **Submitter:** Jens Maurer **Date:** 8 March 2007

[Voted into the WP at the September, 2008 meeting (resolution in paper N2757).]

Issue 256 was closed without action, principally on the the grounds that an implementation could provide a means (command-line option, `#pragma`, etc.) for requesting that the allocation size be checked for validity, but that "it would not be appropriate to require this overhead for every array allocation in every program."

This rationale may be giving too much weight to the overhead such a check would add, especially when compared to the likely cost of actually doing the storage allocation. In particular, the test essentially amounts to something like

```
if (max_allocation_size / sizeof(T) < num_elements)
    throw std::bad_alloc();
```

(noting that `max_allocation_size/sizeof(T)` is a compile-time constant). It might make more sense to turn the rationale around and require the check, assuming that implementations could provide a mechanism for suppressing it if needed.

Suggested resolution:

In 8.3.4 [expr.new] paragraph 7, add the following words before the example:

> If the value of the expression is such that the size of the allocated object would exceed the implementation-defined limit, an exception of type `std::bad_alloc` is thrown and no storage is obtained.

Note (March, 2008):

The Evolution Working Group has accepted the intent of issue 256 and referred it to CWG for action for C++0x (see paper J16/07-0033 = WG21 N2173).

Proposed resolution (March, 2008):

As suggested.

Notes from the June, 2008 meeting:

The CWG felt that this situation should not be treated like an out-of-memory situation and thus an exception of type `std::bad_alloc` (or, alternatively, returning a null pointer for a `throw()` allocator) would not be appropriate.

Proposed resolution (June, 2008):

Change 8.3.4 [expr.new] paragraph 8 as follows:

> **If the value of the *expression* in a *direct-new-declarator* is such that the size of the allocated object would exceed the implementation-defined limit, no storage is obtained and the *new-expression* terminates by throwing an exception of a type that would match a handler (18.3 [except.handle]) of type `std::length_error` (22.2.5 [length.error]). Otherwise, if** ~~When~~ the value of ~~the~~ **that** *expression* ~~in a *direct-new-declarator*~~ is zero, the allocation function is called to allocate an array with no elements.

[Drafting note: `std::length_error` is thrown by `std::string` and `std::vector` and thus appears to be the right choice for the exception to be thrown here.]

---

288. Misuse of "static type" in describing pointers

Section: 8.3.5 [expr.delete] **Status:** CD1 **Submitter:** James Kuyper **Date:** 19 May 2001

[Voted into the WP at the June, 2008 meeting.]

For delete expressions, 8.3.5 [expr.delete] paragraph 1 says

> The operand shall have a pointer type, or a class type having a single conversion function to a pointer type.

However, paragraph 3 of that same section says:

> if the static type of the operand is different from its dynamic type, the static type shall be a base class of the operand's dynamic type and the static type shall have a virtual destructor or the behavior is undefined.

Since the operand must be of pointer type, its static type is necessarily the same as its dynamic type. That clause is clearly referring to the object being pointed at, and not to the pointer operand itself.

Correcting the wording gets a little complicated, because dynamic and static types are attributes of expressions, not objects, and there's no sub-expression of a *delete-expression* which has the relevant types.

Suggested resolution:

> then there is a static type and a dynamic type that the hypothetical expression `(* const-expression)` would have. If that static type is different from that dynamic type, then that static type shall be a base class of that dynamic type, and that static type shall have a virtual destructor, or the behavior is undefined.

There's precedent for such use of hypothetical constructs: see 8.10 [expr.eq] paragraph 2, and 11.1 [dcl.name] paragraph 1.

13.3 [class.virtual] paragraph 3 has a similar problem. It refers to

> the type of the pointer or reference denoting the object (the static type).

The type of the pointer is different from the type of the reference, both of which are different from the static type of '*pointer', which is what I think was actually intended. Paragraph 6 contains the exact same wording, in need of the same correction. In this case, perhaps replacing "pointer or reference" with "expression" would be the best fix. In order for this fix to be sufficient, `pointer->member` must be considered equivalent to `(*pointer).member`, in which case the "expression" referred to would be `(*pointer)`.

15.5 [class.free] paragraph 4 says that

> if a *delete-expression* is used to deallocate a class object whose static type has...

This should be changed to

> if a *delete-expression* is used to deallocate a class object through a pointer expression whose dereferenced static type would have...

The same problem occurs later, when it says that the

> static and dynamic types of the object shall be identical

In this case you could replace "object" with "dereferenced pointer expression".

Footnote 104 says that

> 8.3.5 [expr.delete] requires that ... the static type of the *delete-expression*'s operand be the same as its dynamic type.

This would need to be changed to

> the *delete-expression*'s dereferenced operand

Proposed resolution (December, 2006):

1. Change 8.3.5 [expr.delete] paragraph 3 as follows:

   > In the first alternative (*delete object*), if the static type of the ~~operand~~ **object to be deleted** is different from its dynamic type, the static type shall be a base class of the ~~operand's~~ dynamic type **of the object to be deleted** and the static type shall have a virtual destructor or the behavior is undefined. In the second alternative (*delete array*) if the dynamic type of the object to be deleted differs from its static type, the behavior is undefined.

2. Change the footnote in 15.5 [class.free] paragraph 4 as follows:

   > A similar provision is not needed for the array version of `operator delete` because 8.3.5 [expr.delete] requires that in this situation, the static type of the ~~*delete-expression*'s operand~~ **object to be deleted** be the same as its dynamic type.

3. Change the footnote in 15.5 [class.free] paragraph 5 as follows:

   > If the static type ~~in the *delete-expression*~~ **of the object to be deleted** is different from the dynamic type and the destructor is not virtual the size might be incorrect, but that case is already undefined; see 8.3.5 [expr.delete].

[*Drafting notes:* No change is required for 13.3 [class.virtual] paragraph 7 because "the type of the pointer" includes the pointed-to type. No change is required for 15.5 [class.free] paragraph 4 because "...used to deallocate a class object whose static type..." already refers to the object (and not the operand expression).]

353. Is deallocation routine called if destructor throws exception in delete?

Section: 8.3.5 [expr.delete] **Status:** CD1 **Submitter:** Duane Smith **Date:** 30 April 2002

[Voted into WP at April 2003 meeting.]

In a couple of comp.std.c++ threads, people have asked whether the Standard guarantees that the deallocation function will be called in a delete-expression if the destructor throws an exception. Most/all people have expressed the opinion that the deallocation function must be called in this case, although no one has been able to cite wording in the Standard supporting that view.

```
#include <new.h>
#include <stdio.h>
#include <stdlib.h>

static int flag = 0;

inline
void operator delete(void* p) throw()
{
   if (flag)
        printf("in deallocation function\n");
   free(p);
}

struct S {
    ~S() { throw 0; }
};

void f() {
    try {
        delete new S;
    }
    catch(...) { }
}

int main() {
       flag=1;
       f();
       flag=0;
       return 0;
}
```

Proposed resolution (October 2002):

Add to 8.3.5 [expr.delete] paragraph 7 the highlighted text:

> The *delete-expression* will call a *deallocation function* (6.7.4.2 [basic.stc.dynamic.deallocation]) **[*Note:* The deallocation function is called regardless of whether the destructor for the object or some element of the array throws an exception.]**

---

442. Incorrect use of null pointer constant in description of delete operator

Section: 8.3.5 [expr.delete] **Status:** CD1 **Submitter:** Matthias Hofmann **Date:** 2 Dec 2003

[Voted into WP at October 2005 meeting.]

After some discussion in comp.lang.c++.moderated we came to the conclusion that there seems to be a defect in 8.3.5 [expr.delete]/4, which says:

> The cast-expression in a delete-expression shall be evaluated exactly once. If the delete-expression calls the implementation deallocation function (3.7.3.2), and if the operand of the delete expression is not the null pointer constant, the deallocation function will deallocate the storage referenced by the pointer thus rendering the pointer invalid. [Note: the value of a pointer that refers to deallocated storage is indeterminate.]

In the second sentence, the term "null pointer constant" should be changed to "null pointer". In its present form, the passage claims that the deallocation function will deallocate the storage refered to by a null pointer that did not come from a null pointer constant in the delete expression. Besides, how can the null pointer constant be the operand of a delete expression, as "delete 0" is an error because delete requires a pointer type or a class type having a single conversion function to a pointer type?

See also issue 348.

Proposed resolution:

Change the indicated sentence of 8.3.5 [expr.delete] paragraph 4 as follows:

> If the *delete-expression* calls the implementation deallocation function (6.7.4.2 [basic.stc.dynamic.deallocation]), and if **the value of** the operand of the delete expression is not ~~the~~ **a** null pointer ~~constant~~, the deallocation function will deallocate the storage referenced by the pointer thus rendering the pointer invalid.

Notes from October 2004 meeting:

This wording is superseded by, and this issue will be resolved by, the resolution of issue 348.

Proposed resolution (April, 2005):

This issue is resolved by the resolution of issue 348.

---

659. Alignment of function types

Section: 8.3.6 [expr.alignof] **Status:** CD1 **Submitter:** Alisdair Meredith **Date:** 7 November 2007

[Voted into the WP at the September, 2008 meeting.]

The specification for the `alignof` operator (8.3.6 [expr.alignof]) does not forbid function types as operands, although it probably should.

Proposed resolution (March, 2008):

The issue, as described, is incorrect. The requirement in 8.3.6 [expr.alignof] is for "a complete object type," so a function type is already forbidden. However, the existing text does have a problem in this requirement in that it does not allow a reference type, as anticipated by paragraph 3. Consequently, the proposal is to change 8.3.6 [expr.alignof] paragraph 1 as indicated:

> An `alignof` expression yields the alignment requirement of its operand type. The operand shall be a *type-id* representing a complete object type **or a reference to a complete object type**.

---

520. Old-style casts between incomplete class types

Section: 8.4 [expr.cast] **Status:** CD1 **Submitter:** comp.std.c++ **Date:** 19 May 2005

[Voted into WP at April, 2007 meeting.]

8.4 [expr.cast] paragraph 6 says,

> The operand of a cast using the cast notation can be an rvalue of type "pointer to incomplete class type". The destination type of a cast using the cast notation can be "pointer to incomplete class type". In such cases, even if there is a inheritance relationship between the source and destination classes, whether the `static_cast` or `reinterpret_cast` interpretation is used is unspecified.

The wording seems to allow the following:

1. casting from void pointer to incomplete type

```
struct A;
struct B;

void *v;
A *a = (A*)v; // allowed to choose reinterpret_cast
```

2. variant application of static or reinterpret casting

```
B *b = (B*)a;    // compiler can choose static_cast here
A *aa = (A*)b;   // compiler can choose reinterpret_cast here
assert(aa == a); // might not hold
```

3. ability to somehow choose static_cast

   It's not entirely clear how a compiler can choose `static_cast` as 8.4 [expr.cast] paragraph 6 seems to allow. I believe the intent of 8.4 [expr.cast] paragraph 6 is to force the use of `reinterpret_cast` when either are incomplete class types and `static_cast` iff the compiler knows both types and there is a non-ambiguous hierarchy-traversal between that cast (or maybe not, core issue 242 talks about this). I cannot see any other interpretation because it isn't intuitive, every compiler I've tried agrees with me, and neither standard pointer conversions (7.11 [conv.ptr] paragraph 3) nor `static_cast` (8.2.9 [expr.static.cast] paragraph 5) talk about incomplete class types. If the committee agrees with me, I would like to see 7.11 [conv.ptr] paragraph 3 and 8.2.9 [expr.static.cast] paragraph 5 explicitly disallow incomplete class types and the wording of 8.4 [expr.cast] paragraph 6 changed to not allow any other interpretation.

Proposed resolution (April, 2006):

Change 8.4 [expr.cast] paragraph 6 as indicated:

> The operand of a cast using the cast notation can be an rvalue of type "pointer to incomplete class type." The destination type of a cast using the cast notation can be "pointer to incomplete class type." ~~In such cases, even if there is a inheritance relationship between the source and destination classes, whether the `static_cast` or `reinterpret_cast` interpretation is used is unspecified.~~ **If both the operand and destination types are class types and one or both are incomplete, it is unspecified whether the `static_cast` or the `reinterpret_cast` interpretation is used, even if there is an inheritance relationship between the two classes. [*Note:* For example, if the classes were defined later in the translation unit, a multi-pass compiler would be permitted to interpret a cast between pointers to the classes as if the class types were complete at that point. —*end note*]**

---

497. Missing required initialization in example

Section: 8.5 [expr.mptr.oper] **Status:** CD1 **Submitter:** Giovanni Bajo **Date:** 03 Jan 2005

[Voted into WP at October 2005 meeting.]

8.5 [expr.mptr.oper] paragraph 5 contains the following example:

```
struct S {
    mutable int i;
};
const S cs;
int S::* pm = &S::i;    // pm refers to mutable member S::i
cs.*pm = 88;            // ill-formed: cs is a const object
```

The const object `cs` is not explicitly initialized, and class `S` does not have a user-declared default constructor. This makes the code ill-formed as per 11.6 [dcl.init] paragraph 9.

Proposed resolution (April, 2005):

Change the example in 8.5 [expr.mptr.oper] paragraph 5 to read as follows:

```
struct S {
    S() : i(0) { }
    mutable int i;
};
void f()
 {
    const S cs;
    int S::* pm = &S::i;    // pm refers to mutable member S::i
    cs.*pm = 88;            // ill-formed: cs is a const object
}
```

---

614. Results of integer `/` and `%`

Section: 8.6 [expr.mul] **Status:** CD1 **Submitter:** Gabriel Dos Reis **Date:** 15 January 2007

[Voted into the WP at the September, 2008 meeting as part of paper N2757.]

The current Standard leaves it implementation-defined whether integer division rounds the result toward 0 or toward negative infinity and thus whether the result of `%` may be negative. C99, apparently reflecting (nearly?) unanimous hardware practice, has adopted the rule that integer division rounds toward 0, thus requiring that the result of `-1 % 5` be `-1`. Should the C++ Standard follow suit?

On a related note, does `INT_MIN % -1` invoke undefined behavior? The `%` operator is defined in terms of the `/` operator, and `INT_MIN / -1` overflows, which by 8 [expr] paragraph 5 causes undefined behavior; however, that is not the "result" of the `%` operation, so it's not clear. The wording of 8.6 [expr.mul] paragraph 4 appears to allow `%` to cause undefined behavior only when the second operand is 0.

Proposed resolution (August, 2008):

Change 8.6 [expr.mul] paragraph 4 as follows:

> The binary `/` operator yields the quotient, and the binary `%` operator yields the remainder from the division of the first expression by the second. If the second operand of `/` or `%` is zero the behavior is undefined~~; otherwise `(a/b)*b + a%b` is equal to `a`. If both operands are nonnegative then the remainder is nonnegative; if not, the sign of the remainder is implementation-defined. [*Footnote:* According to work underway toward the revision of ISO C, the preferred algorithm for integer division follows the rules defined in the ISO Fortran standard, ISO/IEC 1539:1991, in which the quotient is always rounded toward zero. —*end footnote*~~]. **For integral operands, the `/` operator yields the algebraic quotient with any fractional part discarded; [*Footnote:* This is often called "truncation towards zero." —*end footnote*] if the quotient `a/b` is representable in the type of the result, `(a/b)*b + a%b` is equal to `a`.**

[Drafting note: see C99 6.5.5 paragraph 6.]

---

661. Semantics of arithmetic comparisons

Section: 8.9 [expr.rel] **Status:** CD1 **Submitter:** Daveed Vandevoorde **Date:** 27 November 2007

[Voted into the WP at the June, 2008 meeting.]

The actual semantics of arithmetic comparison — e.g., whether `1 < 2` yields `true` or `false` — appear not to be specified anywhere in the Standard. The C Standard has a general statement that

> Each of the operators `<` (less than), `>` (greater than), `<=` (less than or equal to), and `>=` (greater than or equal to) shall yield 1 if the specified relation is true and 0 if it is false.

There is no corresponding statement in the C++ Standard.

Proposed resolution (February, 2008):

1. Append the following paragraph to the end of 8.9 [expr.rel]:

   > If both operands (after conversions) are of arithmetic type, each of the operators shall yield `true` if the specified relation is true and `false` if it is false.

2. Append the following paragraph to the end of 8.10 [expr.eq]:

   > Each of the operators shall yield `true` if the specified relation is true and `false` if it is false.

---

446. Does an lvalue-to-rvalue conversion on the "?" operator produce a temporary?

Section: 8.16 [expr.cond] **Status:** CD1 **Submitter:** John Potter **Date:** 31 Dec 2003

[Voted into WP at October 2005 meeting.]

The problem occurs when the value of the operator is determined to be an rvalue, the selected argument is an lvalue, the type is a class type and a non-const member is invoked on the modifiable rvalue result.

```
struct B {
    int v;
    B (int v) : v(v) { }
    void inc () { ++ v; }
    };
struct D : B {
    D (int v) : B(v) { }
    };

B b1(42);
(0 ? B(13) : b1).inc();
assert(b1.v == 42);
```

The types of the second and third operands are the same and one is an rvalue. Nothing changes until p6 where an lvalue to rvalue conversion is performed on the third operand. 15.2 [class.temporary] states that an lvalue to rvalue conversion produces a temporary and there is nothing to remove it. It seems clear that the assertion must pass, yet most implementations fail.

There seems to be a defect in p3 b2 b1. First, the conditions to get here and pass the test.

> If E1 and E2 have class type, and the underlying class types are the same or one is a base class of the other: E1 can be converted to match E2 if the class of T2 is the same type as, or a base class of, the class of T1, and the cv-qualification of T2 is the same cv-qualification as, or a greater cv-qualification than, the cv-qualification of T1.

If both E1 and E2 are lvalues, passing the conditions here also passes the conditions for p3 b1. Thus, at least one is an rvalue. The case of two rvalues is not interesting and the action is covered by the case when E1 is an rvalue.

```
(0 ? D(13) : b1).inc();
assert(b1.v == 42);
```

> E1 is changed to an rvalue of type T2 that still refers to the original source class object (or the appropriate subobject thereof). [Note: that is, no copy is made.]

Having changed the rvalue to base type, we are back to the above case where an lvalue to rvalue conversion is required on the third operand at p6. Again, most implementations fail.

The remaining case, E1 an lvalue and E2 an rvalue, is the defect.

```
D d1(42);
(0 ? B(13) : d1).inc();
assert(d1.v == 42);
```

The above quote states that an lvalue of type T1 is changed to an rvalue of type T2 without creating a temporary. This is in contradiction to everything else in the standard about lvalue to rvalue conversions. Most implementations pass in spite of the defect.

The usual accessible and unambiguous is missing from the base class.

There seems to be two possible solutions. Following other temporary creations would produce a temporary rvalue of type T1 and change it to an rvalue of type T2. Keeping the no copy aspect of this bullet intact would change the lvalue of type T1 to an lvalue of type T2. In this case the lvalue to rvalue conversion would happen in p6 as usual.

Suggested wording for p3 b2 b1

The base part:

> If E1 and E2 have class type, and the underlying class types are the same or one is a base class of the other: E1 can be converted to match E2 if the class of T2 is the same type as, or an accessible and unambiguous base class of, the class of T1, and the cv-qualification of T2 is the same cv-qualification as, or a greater cv-qualification than, the cv-qualification of T1. If the conversion is applied:

The same type temporary version:

If E1 is an lvalue, an lvalue to rvalue conversion is applied. The resulting or original rvalue is changed to an rvalue of type T2 that refers to the same class object (or the appropriate subobject thereof). [Note: that is, no copy is made in changing the type of the rvalue.]

The never copy version:

> The lvalue(rvalue) E1 is changed to an lvalue(rvalue) of type T2 that refers to the original class object (or the appropriate subobject thereof). [Note: that is, no copy is made.]

The test case was posted to clc++m and results for implementations were reported.

```
#include <cassert>
struct B {
    int v;
    B (int v) : v(v) { }
    void inc () { ++ v; }
    };
struct D : B {
    D (int v) : B(v) { }
    };
int main () {
    B b1(42);
    D d1(42);
    (0 ? B(13) : b1).inc();
    assert(b1.v == 42);
    (0 ? D(13) : b1).inc();
    assert(b1.v == 42);
    (0 ? B(13) : d1).inc();
    assert(d1.v == 42);
    }

// CbuilderX(EDG301) FFF  Rob Williscroft
// ICC-8.0           FFF  Alexander Stippler
// COMO-4.301        FFF  Alexander Stippler

// BCC-5.4           FFP  Rob Williscroft
// BCC32-5.5         FFP  John Potter
// BCC32-5.65        FFP  Rob Williscroft
// VC-6.0            FFP  Stephen Howe
// VC-7.0            FFP  Ben Hutchings
// VC-7.1            FFP  Stephen Howe
// OpenWatcom-1.1    FFP  Stephen Howe

// Sun C++-6.2       PFF  Ron Natalie

// GCC-3.2           PFP  John Potter
// GCC-3.3           PFP  Alexander Stippler

// GCC-2.95          PPP  Ben Hutchings
// GCC-3.4           PPP  Florian Weimer
```

I see no defect with regards to lvalue to rvalue conversions; however, there seems to be disagreement about what it means by implementers. It may not be surprising because 5.16 and passing a POD struct to an ellipsis are the only places where an lvalue to rvalue conversion applies to a class type. Most lvalue to rvalue conversions are on basic types as operands of builtin operators.

Notes from the March 2004 meeting:

We decided all "?" operators that return a class rvalue should copy the second or third operand to a temporary. See issue 86.

Proposed resolution (October 2004):

1. Change 8.16 [expr.cond] paragraph 3 bullet 2 sub-bullet 1 as follows:

   > if `E1` and `E2` have class type, and the underlying class types are the same or one is a base class of the other: `E1` can be converted to match `E2` if the class of `T2` is the same type as, or a base class of, the class of `T1`, and the cv-qualification of `T2` is the same cv-qualification as, or a greater cv-qualification than, the cv-qualification of `T1`. If the conversion is applied, `E1` is changed to an rvalue of type `T2` ~~that still refers to the original source class object (or the appropriate subobject thereof). [*Note:* that is, no copy is made. —*end note*]~~ **by copy-initializing a temporary of type `T2` from `E1` and using that temporary as the converted operand.**

2. Change 8.16 [expr.cond] paragraph 6 bullet 1 as follows:

   > The second and third operands have the same type; the result is of that type. **If the operands have class type, the result is an rvalue temporary of the result type, which is copy-initialized from either the second operand or the third operand depending on the value of the first operand.**

3. Change 7.1 [conv.lval] paragraph 2 as follows:

   > ~~The value contained in the object indicated by the lvalue is the rvalue result.~~ When an lvalue-to-rvalue conversion occurs within the operand of `sizeof` (8.3.3 [expr.sizeof]) the value contained in the referenced object is not accessed, since that operator does not evaluate its operand. **Otherwise, if the lvalue has a class type, the conversion copy-initializes a temporary of type `T` from the lvalue and the result of the conversion is an rvalue for the temporary. Otherwise, the value contained in the object indicated by the lvalue is the rvalue result.**

[Note: this wording partially resolves issue 86. See also issue 462.]

339. Overload resolution in operand of `sizeof` in constant expression

Section: 8.20 [expr.const] **Status:** CD1 **Submitter:** Steve Adamczyk **Date:** 11 Mar 2002

[Voted into the WP at the June, 2008 meeting as paper N2634.]

I've seen some pieces of code recently that put complex expressions involving overload resolution inside `sizeof` operations in constant expressions in templates.

8.20 [expr.const] paragraph 1 implies that some kinds of nonconstant expressions are allowed inside a `sizeof` in a constant expression, but it's not clear that this was intended to extend all the way to things like overload resolution. Allowing such things has some hidden costs. For example, name mangling has to be able to represent all operators, including calls, and not just the operators that can appear in constant expressions.

```
template <int I> struct A {};

char xxx(int);
char xxx(float);

template <class T> A<sizeof(xxx((T)0))> f(T){}

int main()
{
  f(1);
}
```

If complex expressions are indeed allowed, it should be because of an explicit committee decision rather than because of some looseness in this section of the standard.

Notes from the 4/02 meeting:

Any argument for restricting such expressions must involve a cost/benefit ratio: a restriction would be palatable only if it causes minimum hardship for users and allows a substantial reduction in implementation cost. If we propose a restriction, it must be one that library writers can live with.

Lots of these cases fail with current compilers, so there can't be a lot of existing code using them. We plan to find out what cases there are in libraries like Loki and Boost.

We noted that in many cases one can move the code into a class to get the same result. The implementation problem comes up when the expression-in-sizeof is in a template deduction context or part of a template signature. The problem cases are ones where an error causes deduction to fail, as opposed to contexts where an error causes a diagnostic. The latter contexts are easier to handle; however, there are situations where "fail deduction" may be the desired behavior.

Notes from the April 2003 meeting:

Here is a better example:

```
extern "C" int printf(const char *, ...);
char f(int);
int f(...);
// Approach 1 -- overload resolution in template class
// No problem
template <class T> struct conv_int {
  static const bool value = (sizeof(f(T())) == 1);
};
// Approach 2 -- overload resolution in type deduction
// Difficult
template <int I> struct A {
  static const int value = I;
};
template <class T> bool conv_int2(A<sizeof(f(T()))> p) {
  return p.value == 1;
}

template<typename T>
A<sizeof(f(T()))> make_A() {
  return A<sizeof(f(T()))>();
}

int main() {
  printf("short: %d\n", conv_int<short>::value);
  printf("int *: %d\n", conv_int<int *>::value);
  printf("short: %d\n", conv_int2<short>(make_A<short>()));
  printf("int *: %d\n", conv_int2<int *>(make_A<int*>()));
}
```

The core working group liked the idea of a restriction that says that expressions inside sizeof in template signature contexts must be otherwise valid as nontype template argument expressions (i.e., integer operations only, limited casts). This of course is subject to whether users can live with that restriction. This topic was brought up in full committee, but there was limited feedback from other groups.

It was also noted that if typeof (whatever it is called) is added, there may be a similar issue there.

Note (March, 2005):

Dave Abrahams (quoting a Usenet posting by Vladimir Marko): The decltype and auto proposal (revision 3: N1607) presents

```
  template <class T,class U>
  decltype((*(T*)0)+(*(U*)0)) add(const T& t,const U& u);
```

as a valid declaration (if the proposal is accepted). If [the restrictions in the April, 2003 note] really applied to decltype, the declaration above would be invalid. AFAICT every non-trivial use of decltype in a template function declaration would be invalid. And for me this would render my favorite proposal useless.

I would propose to allow any kind of expression inside `sizeof` (and `decltype`) and explicitly add `sizeof` (and `decltype`) expressions involving template-parameters to non-deduced contexts (add a bullet to 17.9.2.4 [temp.deduct.partial] paragraph 4).

Jaakko Jarvi: Just reinforcing that this is important and hope for insights. The topic is discussed a bit on page 10 of the latest revision of the proposal (N1705). Here's a quote from the proposal:

> However, it is crucial that no restrictions are placed on what kinds of expressions are allowed inside ***decltype***, and therefore also inside ***sizeof***. We suggest that issue 339 is resolved to require the compiler to fail deduction (apply the SFINAE principle), and not produce an error, for as large set of invalid expressions in operands of ***sizeof*** or ***decltype*** as is possible to comfortably implement. We wish that implementors aid in classifying the kinds of expressions that should produce errors, and the kinds that should lead to failure of deduction.

Notes from the April, 2007 meeting:

The CWG is pursuing a compromise proposal, to which the EWG has tentatively agreed, which would allow arbitrary expressions in the return types of function templates but which would restrict the expressions that participate in the function signature (and thus in overload resolution) to those that can be used as non-type template arguments. During deduction and overload resolution, these complex return types would be ignored; that is, there would be no substitution of the deduced template arguments into the return type at this point. If such a function were selected by overload resolution, however, a substitution failure in the return type would produce a diagnostic rather than a deduction failure.

This approach works when doing overload resolution in the context of a function call, but additional tricks (still being defined) are needed in other contexts such as friend function declaration matching and taking the address of a function, in which the return type does play a part.

Notes from the July, 2007 meeting:

The problem is whether arbitrary expressions (for example, ones that include overload resolution) are allowed in template deduction contexts, and, if so, which expression errors are SFINAE failures and which are hard errors.

This issue deals with arbitrary expressions inside sizeof in deduction contexts. That's a fringe case right now (most compilers don't accept them). decltype makes the problem worse, because the standard use case is one that involves overload resolution. Generalized constant expressions make it worse yet, because they allow overload resolution and class types to show up in any constant expression in a deduction context.

Why is this an issue? Why don't we just say everything is allowed and be done with it?

- Because it's hard to implement the general case. Template deduction/substitution has historically used a simplified model of semantic checking, i.e., the SFINAE rules (which are mostly about types), instead of full semantic checking. This limited semantic checking is typically done by completely separate code from the code for "normal" expression checking, and it's not easy to extend it to the general case. "Speculative compilation" sounds like an easy way out, but in practice compilers can't do that.
- Because it affects name mangling and therefore the ABI.
- Because we need to figure out what to say and how to say it in the Standard.

At the April, 2007 meeting, we were headed toward a solution that imposed a restriction on expressions in deduction contexts, but such a restriction seems to really hamper uses of constexpr functions. So we're now proposing that fully general expressions be allowed, and that most errors in such expressions be treated as SFINAE failures rather than errors.

One issue with writing Standard wording for that is how to define "most." There's a continuum of errors, some errors being clearly SFINAE failures, and some clearly "real" errors, with lots of unclear cases in between. We decided it's easier to write the definition by listing the errors that are not treated as SFINAE failures, and the list we came up with is as follows:

1. errors that occur while processing some entity external to the expression, e.g., an instantiation of a template or the generation of the definition of an implicitly-declared copy constructor
2. errors due to implementation limits
3. errors due to access violations (this is a judgment call, but the philosophy of access has always been that it doesn't affect visibility)

Everything else produces a SFINAE failure rather than a hard error.

There was broad consensus that this felt like a good solution, but that feeling was mixed with trepidation on several fronts:

- The implementation cost is quite significant, at least for EDG and Microsoft (under GCC, it may be easier). It involves moving around a large amount of code. This may delay implementation and introduce bugs in compilers.
- While it seems upward compatible with C++03, it's possible it will break existing code. Any big change in template processing has a pretty good chance of breaking something.
- Since there is no implementation, we don't really know how it will work in the real world.

We will be producing wording for the Working Draft for the October, 2007 meeting.

(See also issue 657.)

366. String literal allowed in integral constant expression?

Section: 8.20 [expr.const] **Status:** CD1 **Submitter:** Martin v. Loewis **Date:** 29 July 2002

[Voted into WP at October 2003 meeting.]

According to 19.1 [cpp.cond] paragraph 1, the if-group

```
#if "Hello, world"
```

is well-formed, since it is an integral constant expression. Since that may not be obvious, here is why:

8.20 [expr.const] paragraph 1 says that an integral constant expression may involve literals (5.13 [lex.literal]); "Hello, world" is a literal. It restricts operators to not use certain type conversions; this expression does not use type conversions. It further disallows functions, class objects, pointers, ... - this expression is none of those, since it is an array.

However, 19.1 [cpp.cond] paragraph 6 does not explain what to do with this if-group, since the expression evaluates neither to false(zero) nor true(non-zero).

Proposed resolution (October 2002):

Change the beginning of the second sentence of 8.20 [expr.const] paragraph 1 which currently reads

> An *integral constant-expression* can involve only literals (5.13 [lex.literal]), ...

to say

> An *integral constant-expression* can involve only literals of arithmetic types (5.13 [lex.literal], 6.9.1 [basic.fundamental]), ...

---

367. `throw` **operator allowed in constant expression?**

Section: 8.20 [expr.const] **Status:** CD1 **Submitter:** Martin v. Loewis **Date:** 29 July 2002

[Voted into WP at the October, 2006 meeting.]

The following translation unit appears to be well-formed.

```
int x[true?throw 4:5];
```

According to 8.20 [expr.const], this appears to be an integral constant expression: it is a conditional expression, involves only literals, and no assignment, increment, decrement, function-call, or comma operators. However, if this is well-formed, the standard gives no meaning to this declaration, since the array bound (11.3.4 [dcl.array] paragraph 1) cannot be computed.

I believe the defect is that throw expressions should also be banned from constant expressions.

Notes from October 2002 meeting:

We should also check on `new` and `delete`.

Notes from the April, 2005 meeting:

Although it could be argued that all three of these operators potentially involve function calls — `throw` to `std::terminate`, `new` and `delete` to the corresponding allocation and deallocation functions — and thus would already be excluded from constant expressions, this reasoning was considered to be too subtle to allow closing the issue with no change. A modification that explicitly clarifies the status of these operators will be drafted.

Proposed resolution (October, 2005):

Change the last sentence of 8.20 [expr.const] as indicated:

> In particular, except in `sizeof` expressions, functions, class objects, pointers, or references shall not be used, and assignment, increment, decrement, ~~function call~~ **function call (including *new-expression*s and *delete-expression*s)**, ~~or~~ comma operators**, or *throw-expression*s** shall not be used.

Note: this sentence is also changed by the resolution of issue 530.

---

457. Wording nit on use of const variables in constant expressions

Section: 8.20 [expr.const] **Status:** CD1 **Submitter:** Mark Mitchell **Date:** 03 Feb 2004

[Voted into WP at April 2005 meeting.]

I'm looking at 8.20 [expr.const]. I see:

> An *integral constant-expression* can involve only ... const variables or static data members of integral or enumeration types initialized with constant expressions ...

Shouldn't that be "const non-volatile"?

It seems weird to say that:

```
const volatile int i = 3;
int j[i];
```

is valid.

Steve Adamczyk: See issue 76, which made the similar change to 10.1.7.1 [dcl.type.cv] paragraph 2, and probably should have changed this one as well.

Proposed resolution (October, 2004):

Change the first sentence in the second part of 8.20 [expr.const] paragraph 1 as follows:

> An *integral constant-expression* can involve only literals of arithmetic types (5.13 [lex.literal], 6.9.1 [basic.fundamental]), enumerators, **non-volatile** `const` variables or static data members of integral or enumeration types initialized with constant expressions (11.6 [dcl.init]), non-type template parameters of integral or enumeration types, and `sizeof` expressions.

---

530. Nontype template arguments in constant expressions

Section: 8.20 [expr.const] **Status:** CD1 **Submitter:** Mark Mitchell **Date:** 21 August 2005

[Voted into the WP at the April, 2007 meeting as part of paper J16/07-0095 = WG21 N2235.]

Consider:

```
template <int* p> struct S {
    static const int I = 3;
};
int i;
int a[S<&i>::I];
```

Clearly this should be valid, but a pedantic reading of 8.20 [expr.const] would suggest that this is invalid because "`&i`" is not permitted in integral constant expressions.

Proposed resolution (October, 2005):

Change the last sentence of 8.20 [expr.const] paragraph 1 as indicated:

> In particular, except in **non-type *template-argument*s or** `sizeof` expressions, functions, class objects, pointers, or references shall not be used, and assignment, increment, decrement, function-call, or comma operators shall not be used.

(Note: the same text is changed by the resolution of issue 367.)

Notes from April, 2006 meeting:

The proposed resolution could potentially be read as saying that the prohibited operations and operators would be permitted in integral constant expressions that are non-type *template-argument*s. John Spicer is investigating an alternate approach, to say that expressions in non-type template arguments are not part of the expression in which the *template-id* appears (in contrast to the operand of `sizeof`, which is part of the containing expression).

Additional note (May, 2008):

This issue is resolved by the rewrite of 8.20 [expr.const] that was done in conjunction with the constexpr proposal, paper N2235.

---

684. Constant expressions involving the address of an automatic variable

Section: 8.20 [expr.const] **Status:** CD1 **Submitter:** Jens Maurer **Date:** 13 March, 2008

[Voted into the WP at the September, 2008 meeting (resolution in paper N2757).]

The expressions that are excluded from being constant expressions in 8.20 [expr.const] paragraph 2 does not address an example like the following:

```
void f() {
   int a;
   constexpr int* p = &a;     // should be ill-formed, currently isn't
}
```

Suggested resolution:

Add the following bullet to the list in 8.20 [expr.const] paragraph 2:

- an *id-expression* that refers to a variable with automatic storage duration unless a permitted lvalue-to-rvalue conversion is applied (see above)

Proposed resolution (June, 2008):

1. Change 6.6.2 [basic.start.static] paragraph 1 as follows:

   > Objects with static storage duration (6.7.1 [basic.stc.static]) or thread storage duration (3.7.2) shall be zero-initialized (11.6 [dcl.init]) before any other initialization takes place. ~~A reference with static or thread storage duration and an object of trivial or literal type with static or thread storage duration can be initialized with a constant expression (8.20 [expr.const]); this is called *constant initialization.*~~ ***Constant initialization*** **is performed:**
   >
   > - **if an object of trivial or literal type with static or thread storage duration is initialized with a constant expression (8.20 [expr.const]), or**
   > - **if a reference with static or thread storage duration is initialized with a constant expression that is not an lvalue designating an object with thread or automatic storage duration.**
   >
   > Together, zero-initialization and constant initialization...

2. Add the following in 8.20 [expr.const] paragraph 2:

   - an lvalue-to-rvalue conversion (4.1) unless it is applied to...
   - **an array-to-pointer conversion (7.2 [conv.array]) that is applied to an lvalue that designates an object with thread or automatic storage duration**
   - **a unary operator `&` (8.3.1 [expr.unary.op]) that is applied to an lvalue that designates an object with thread or automatic storage duration**
   - an *id-expression* that refers to a variable or data member of reference type;
   - ...

(Note: the change to 6.6.2 [basic.start.static] paragraph 1 needs to be reconciled with the conflicting change in issue 688.)

---

276. Order of destruction of parameters and temporaries

Section: 9.6 [stmt.jump] **Status:** CD1 **Submitter:** James Kanze **Date:** 28 Mar 2001

[Voted into the WP at the June, 2008 meeting.]

According to 9.6 [stmt.jump] paragraph 2,

> On exit from a scope (however accomplished), destructors (15.4 [class.dtor]) are called for all constructed objects with automatic storage duration (6.7.3 [basic.stc.auto]) (named objects or temporaries) that are declared in that scope, in the reverse order of their declaration.

This wording is problematic for temporaries and for parameters. First, temporaries are not "declared," so this requirement does not apply to them, in spite of the assertion in the quoted text that it does.

Second, although the parameters of a function are *declared* in the called function, they are constructed and destroyed in the calling context, and the order of evaluation of the arguments is unspecified (cf 8.2.2 [expr.call] paragraphs 4 and 8). The order of destruction of the parameters might, therefore, be different from the reverse order of their declaration.

Notes from 04/01 meeting:

Any resolution of this issue should be careful not to introduce requirements that are redundant or in conflict with those of other parts of the IS. This is especially true in light of the pending issues with respect to the destruction of temporaries (see issues 86, 124, 199, and 201). If possible, the wording of a resolution should simply reference the relevant sections.

It was also noted that the temporary for a return value is also destroyed "out of order."

Note that issue 378 picks a nit with the wording of this same paragraph.

Proposed Resolution (November, 2006):

Change 9.6 [stmt.jump] paragraph 2 as follows:

> On exit from a scope (however accomplished), ~~destructors (15.4 [class.dtor]) are called for all constructed objects with automatic storage duration (6.7.3 [basic.stc.auto]) (named objects or temporaries) that are declared in that scope, in the reverse order of their declaration.~~ **variables with automatic storage duration (6.7.3 [basic.stc.auto]) that have been constructed in that scope are destroyed in the reverse order of their construction. [*Note:* For temporaries, see 15.2 [class.temporary]. — *end note*]** Transfer out of a loop...

---

378. Wording that says temporaries are declared

Section: 9.6 [stmt.jump] **Status:** CD1 **Submitter:** Gennaro Prota **Date:** 07 September 2002

Paragraph 9.6 [stmt.jump] paragraph 2 of the standard says:

> On exit from a scope (however accomplished), destructors (15.4 [class.dtor]) are called for all constructed objects with automatic storage duration (6.7.3 [basic.stc.auto]) (named objects or temporaries) that are declared in that scope.

It refers to objects "that are declared" but the text in parenthesis also mentions temporaries, which cannot be declared. I think that text should be removed.

This is related to issue 276.

Proposed Resolution (November, 2006):

This issue is resolved by the resolution of issue 276.

---

281. `inline` **specifier in** `friend` **declarations**

Section: 10.1.2 [dcl.fct.spec] **Status:** CD1 **Submitter:** John Spicer **Date:** 24 Apr 2001

[Moved to DR at October 2002 meeting.]

There is currently no restriction on the use of the `inline` specifier in `friend` declarations. That would mean that the following usage is permitted:

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

struct B {
    friend inline void A::f();
};

void A::f(){}
```

I believe this should be disallowed because a `friend` declaration in one class should not be able to change attributes of a member function of another class.

More generally, I think that the `inline` attribute should only be permitted in `friend` declarations that are definitions.

Notes from the 04/01 meeting:

The consensus agreed with the suggested resolution. This outcome would be similar to the resolution of issue 136.

Proposed resolution (10/01):

Add to the end of 10.1.2 [dcl.fct.spec] paragraph 3:

> If the `inline` specifier is used in a friend declaration, that declaration shall be a definition or the function shall have previously been declared inline.

---

317. Can a function be declared inline after it has been called?

Section: 10.1.2 [dcl.fct.spec] **Status:** CD1 **Submitter:** Steve Clamage **Date:** 14 Oct 2001

[Voted into WP at October 2005 meeting.]

Steve Clamage: Consider this sequence of declarations:

```
void foo() { ... }
inline void foo();
```

The non-inline definition of foo precedes the inline declaration. It seems to me this code should be ill-formed, but I could not find anything in the standard to cover the situation.

Bjarne Stroustrup: Neither could I, so I looked in the ARM, which addressed this case (apparently for member function only) in some detail in 7.1.2 (pp103-104).

The ARM allows declaring a function inline after its initial declaration, as long as it has not been called.

Steve Clamage: If the above code is valid, how about this:

```
void foo() { ... }    // define foo
void bar() { foo(); } // use foo
inline void foo();    // declare foo inline
```

Bjarne Stroustrup: ... and [the ARM] disallows declaring a function inline after it has been called.

This may still be a good resolution.

Steve Clamage: But the situation in the ARM is the reverse: Declare a function inline, and define it later (with no intervening call). That's a long-standing rule in C++, and allows you to write member function definitions outside the class.

In my example, the compiler could reasonably process the entire function as out-of-line, and not discover the inline declaration until it was too late to save the information necessary for inline generation. The equivalent of another compiler pass would be needed to handle this situation.

Bjarne Stroustrup: I see, and I think your argument it conclusive.

Steve Clamage: I'd like to open a core issue on this point, and I recommend wording along the lines of: "A function defined without an inline specifier shall not be followed by a declaration having an inline specifier."

I'd still like to allow the common idiom

```
class T {
  int f();
};
inline int T::f() { ... }
```

Martin Sebor: Since the inline keyword is just a hint to the compiler, I don't see any harm in allowing the construct. Your hypothetical compiler can simply ignore the inline on the second declaration. On the other hand, I feel that adding another special rule will unnecessarily complicate the language.

Steve Clamage: The inline specifier is more than a hint. You can have multiple definitions of inline functions, but only one definition of a function not declared inline. In particular, suppose the above example were in a header file, and included multiple times in a program.

Proposed resolution (October, 2004):

Add the indicated words to 10.1.2 [dcl.fct.spec] paragraph 4:

> An inline function shall be defined in every translation unit in which it is used and shall have exactly the same definition in every case (6.2 [basic.def.odr]). [*Note:* a call to the inline function may be encountered before its definition appears in the translation unit. *—end note*] **If the definition of a function appears in a translation unit before its first declaration as inline, the program is ill-formed.** If a function with external linkage is declared inline in one translation unit...

---

396. Misleading note regarding use of `auto` for disambiguation

Section: 10.1.2 [dcl.fct.spec] **Status:** CD1 **Submitter:** Herb Sutter **Date:** 3 Jan 2003

[Voted into WP at March 2004 meeting.]

BTW, I noticed that the following note in 10.1.1 [dcl.stc] paragraph 2 doesn't seem to have made it onto the issues list or into the TR:

> [Note: hence, the auto specifier is almost always redundant and not often used; one use of auto is to distinguish a declaration-statement from an expression-statement (stmt.ambig) explicitly. --- end note]

I thought that this was well known to be incorrect, because using auto does not disambiguate this. Writing:

```
auto int f();
```

is still a declaration of a function f, just now with an error since the function's return type may not use an auto storage class specifier. I suppose an error is an improvement over a silent ambiguity going the wrong way, but it's still not a solution for the user who wants to express the other in a compilable way.

Proposed resolution: Replace that note with the following note:

> [Note: hence, the auto specifier is always redundant and not often used. --- end note]

John Spicer: I support the proposed change, but I think the disambiguation case is not the one that you describe. An example of the supposed disambiguation is:

```
int i;
int j;
int main()
{
  int(i);  // declares i, not reference to ::i
  auto int(j);  // declares j, not reference to ::j
}
```

cfront would take "int(i)" as a cast of ::i, so the auto would force what it would otherwise treat as a statement to be considered a declaration (cfront 3.0 warned that this would change in the future).

In a conforming compiler the auto is always redundant (as you say) because anything that could be considered a valid declaration should be treated as one.

Proposed resolution (April 2003):

Replace 10.1.1 [dcl.stc] paragraph 2

> [Note: hence, the `auto` specifier is almost always redundant and not often used; one use of `auto` is to distinguish a *declaration-statement* from an *expression-statement* (9.8 [stmt.ambig]) explicitly. --- end note]

with

> [Note: hence, the `auto` specifier is always redundant and not often used. One use of `auto` is to distinguish a *declaration-statement* from an *expression-statement* explicitly rather than relying on the disambiguation rules (9.8 [stmt.ambig]), which may aid readers. --- end note]

---

397. Same address for string literals from default arguments in inline functions?

Section: 10.1.2 [dcl.fct.spec] **Status:** CD1 **Submitter:** Mark Mitchell **Date:** 13 Jan 2003

[Voted into WP at April, 2007 meeting.]

Are string literals from default arguments used in extern inlines supposed to have the same addresses across all translation units?

```
void f(const char* = "s")
inline g() {
  f();
}
```

Must the "s" strings be the same in all copies of the inline function?

Steve Adamczyk: The totality of the standard's wisdom on this topic is (10.1.2 [dcl.fct.spec] paragraph 4):

> A string literal in an extern inline function is the same object in different translation units.

I'd hazard a guess that a literal in a default argument expression is not "in" the extern inline function (it doesn't appear in the tokens of the function), and therefore it need not be the same in different translation units.

I don't know that users would expect such strings to have the same address, and an equally valid (and incompatible) expectation would be that the same string literal would be used for every expansion of a given default argument in a single translation unit.

Notes from April 2003 meeting:

The core working group feels that the address of a string literal should be guaranteed to be the same only if it actually appears textually within the body of the inline function. So a string in a default argument expression in a block extern declaration inside the body of a function would be the same in all instances of the function. On the other hand, a string in a default argument expression in the header of the function (i.e., outside of the body) would not be the same.

Proposed resolution (April 2003):

Change the last sentence and add the note to the end of 10.1.2 [dcl.fct.spec] paragraph 4:

> A string literal in **the body of** an `extern inline` function is the same object in different translation units. **[*Note:* A string literal that is encountered only in the context of a function call (in the default argument expression of the called function), is not "in" the `extern inline` function.]**

Notes from October 2003 meeting:

We discussed ctor-initializer lists and decided that they are also part of the body. We've asked Clark Nelson to work on syntax changes to give us a syntax term for the body of a function so we can refer to it here. See also issue 452, which could use this term.

(October, 2005: moved to "review" status in concert with issue 452. With that resolution, the wording above needs no further changes.)

Proposed resolution (April, 2006):

Change the last sentence and add the note to the end of 10.1.2 [dcl.fct.spec] paragraph 4:

> A string literal in **the body of** an `extern inline` function is the same object in different translation units. **[*Note:* A string literal appearing in a default argument expression is not considered to be "in the body" of an inline function merely by virtue of the expression's use in a function call from that inline function. —*end note*]**

---

477. Can `virtual` appear in a `friend` declaration?

Section: 10.1.2 [dcl.fct.spec] **Status:** CD1 **Submitter:** Daveed Vandevoorde **Date:** 23 Sep 2004

[Voted into WP at the October, 2006 meeting.]

I couldn't find wording that makes it invalid to say `friend virtual...` The closest seems to be 10.1.2 [dcl.fct.spec] paragraph 5, which says:

> The `virtual` specifier shall only be used in declarations of nonstatic class member functions that appear within a *member-specification* of a class definition; see 13.3 [class.virtual].

I don't think that excludes a friend declaration (which is a valid *member-specification* by 12.2 [class.mem]).

John Spicer: I agree that virtual should not be allowed on friend declarations. I think the wording in 10.1.2 [dcl.fct.spec] is intended to be *the* declaration of a function within its class, although I think the wording should be improved to make it clearer.

Proposed resolution (October, 2005):

Change 10.1.2 [dcl.fct.spec] paragraphs 5-6 as indicated:

> The `virtual` specifier shall ~~only~~ be used **only** in ~~declarations~~ **the initial declaration** of **a** non-static class member ~~functions that appear within a *member-specification* of a class definition~~ **function**; see 13.3 [class.virtual].
>
> The `explicit` specifier shall be used only in ~~declarations~~ **the declaration** of ~~constructors~~ **a constructor** within ~~a~~ **its** class definition; see 15.3.1 [class.conv.ctor].

---

424. Wording problem with issue 56 resolution on redeclaring typedefs in class scope

Section: 10.1.3 [dcl.typedef] **Status:** CD1 **Submitter:** Daveed Vandevoorde **Date:** 25 June 2003

[Voted into WP at March 2004 meeting.]

I wonder if perhaps the core issue 56 change in 10.1.3 [dcl.typedef] paragraph 2 wasn't quite careful enough. The intent was to remove the allowance for:

```
struct S {
  typedef int I;
  typedef int I;
};
```

but I think it also disallows the following:

```
class B {
  typedef struct A {} A;
  void f(struct B::A*p);
};
```

See also issue 407.

Proposed resolution (October 2003):

At the end of 10.1.3 [dcl.typedef] paragraph 2, add the following:

> In a given class scope, a `typedef` specifier can be used to redefine any *class-name* declared in that scope that is not also a *typedef-name* to refer to the type to which it already refers. [*Example:*
>
> ```
> struct S {
>   typedef struct A {} A;  // OK
>   typedef struct B B;     // OK
>   typedef A A;            // error
> };
> ```
>
>]

---

647. Non-constexpr instances of constexpr constructor templates

Section: 10.1.5 [dcl.constexpr] **Status:** CD1 **Submitter:** Mike Miller **Date:** 12 Aug 2007

[Voted into the WP at the September, 2008 meeting.]

According to 10.1.5 [dcl.constexpr] paragraph 5,

> If the instantiated template specialization of a constexpr function template would fail to satisfy the requirements for a constexpr function, the `constexpr` specifier is ignored and the specialization is not a constexpr function.

One would expect to see a similar provision for an instantiated constructor template (because the requirements for a constexpr function [paragraph 3] are different from the requirements for a constexpr constructor [paragraph 4]), but there is none; constexpr constructor templates are not mentioned.

Suggested resolution:

Change the wording of 10.1.5 [dcl.constexpr] paragraph 5 as indicated:

> If the instantiated template specialization of a constexpr function template would fail to satisfy the requirements for a constexpr function **or constexpr constructor, as appropriate to the function template**, the `constexpr` specifier is ignored and

the specialization is not a constexpr function **or constexpr constructor**.

Proposed resolution (June, 2008):

[Drafting note: This resolution goes beyond the problem described in the issue discussion, which is one aspect of the general failure of the existing wording to deal consistently with the distinctions between constexpr functions and constexpr constructors. The wording below attempts to rectify that problem systematically.]

1. Change 10.1.5 [dcl.constexpr] paragraph 2 as follows:

   A `constexpr` specifier used in ~~a function declaration~~ **the declaration of a function that is not a constructor** declares that function to be a *constexpr function*. Similarly, a `constexpr` specifier used in a constructor declaration declares that constructor to be a *constexpr constructor*. Constexpr functions and constexpr constructors are implicitly `inline` (10.1.2 [dcl.fct.spec]). ~~A constexpr function shall not be virtual (10.3).~~

2. Change 10.1.5 [dcl.constexpr] paragraph 3 as follows:

   The definition of a constexpr function shall satisfy the following constraints:

   - **it shall not be virtual (13.3 [class.virtual])**
   - its return type shall be a literal type
   - each of its parameter types shall be a literal type
   - its function-body shall be a compound-statement of the form

     `{ return` *expression* `; }`

     where *expression* is a potential constant expression (8.20 [expr.const])
   - every implicit conversion used in converting *expression* to the function return type (11.6 [dcl.init]) shall be one of those allowed in a constant expression (8.20 [expr.const]).

   [*Example:*...

3. Change 10.1.5 [dcl.constexpr] paragraph 4 as follows:

   The definition of a constexpr constructor shall satisfy the following constraints:

   - **each of its parameter types shall be a literal type**
   - **its *function-body* shall not be a *function-try-block***
   - the *compound-statement* of its *function-body* shall be empty
   - every non-static data member and base class sub-object shall be initialized (15.6.2 [class.base.init])
   - every constructor involved in initializing non-static data members and base class sub-objects **invoked by a *mem-initializer*** shall be a constexpr constructor ~~invoked with potential constant expression arguments, if any.~~
   - **every constructor argument and full-expression in a *mem-initializer* shall be a potential constant expression**
   - **every implicit conversion used in converting a constructor argument to the corresponding parameter type and converting a full-expression to the corresponding member type shall be one of those allowed in a constant expression.**

   A trivial copy constructor is also a constexpr constructor. [*Example:* ...

4. Change 10.1.5 [dcl.constexpr] paragraph 5 as follows:

   If the instantiated template specialization of a constexpr function template would fail to satisfy the requirements for a constexpr function **or constexpr constructor**, the `constexpr` specifier is ignored ~~and the specialization is not a constexpr function~~.

5. Change 10.1.5 [dcl.constexpr] paragraph 6 as follows:

   A `constexpr` specifier ~~used in~~ **for** a non-static member function ~~definition~~ **that is not a constructor** declares that member function to be `const` (12.2.2 [class.mfct.non-static]). [*Note:* ...

---

648. Constant expressions in constexpr initializers

Section: 10.1.5 [dcl.constexpr] **Status:** CD1 **Submitter:** Mike Miller **Date:** 12 Aug 2007

[Voted into the WP at the September, 2008 meeting.]

The current wording of 10.1.5 [dcl.constexpr] paragraph 7 seems not quite correct. It reads,

> A `constexpr` specifier used in an object declaration declares the object as `const`. Such an object shall be initialized, and every expression that appears in its initializer (11.6 [dcl.init]) shall be a constant expression.

The phrase "every expression" is intended to cover multiple arguments to a constexpr constructor and multiple expressions in an aggregate initializer. However, it could be read (incorrectly) as saying that non-constant expressions cannot appear as subexpressions in such initializers, even in places where they do not render the full-expression non-constant (i.e., as unevaluated operands and in the unselected branches of `&&`, `||`, and `?:`). Perhaps this problem could be remedied by replacing "every expression" with "every full-expression?"

Proposed resolution (June, 2008):

Change 10.1.5 [dcl.constexpr] paragraph 7 as follows:

> A `constexpr` specifier used in an object declaration declares the object as `const`. Such an object shall be ~~initialized, and every expression that appears in its initializer (8.5)~~ **initialized. If it is initialized by a constructor call, the constructor shall be a constexpr constructor and every argument to the constructor shall be a constant expression. Otherwise, every full-expression that appears in its initializer** shall be a constant expression. Every implicit conversion used...

---

283. Template *type-parameter*s are not syntactically *type-name*s

Section: 10.1.7.2 [dcl.type.simple] **Status:** CD1 **Submitter:** Clark Nelson **Date:** 01 May 2001

[Voted into WP at April 2003 meeting.]

Although 17.1 [temp.param] paragraph 3 contains an assertion that

> A *type-parameter* defines its identifier to be a *type-name* (if declared with `class` or `typename`)

the grammar in 10.1.7.2 [dcl.type.simple] paragraph 1 says that a *type-name* is either a *class-name*, an *enum-name*, or a *typedef-name*. The *identifier* in a template *type-parameter* is none of those. One possibility might be to equate the *identifier* with a *typedef-name* instead of directly with a *type-name*, which would have the advantage of not requiring parallel treatment of the two in situations where they are treated the same (e.g., in *elaborated-type-specifier*s, see issue 245). See also issue 215.

Proposed resolution (Clark Nelson, March 2002):

In 17.1 [temp.param] paragraph 3, change "A *type-parameter* defines its identifier to be a *type-name*" to "A *type-parameter* defines its identifier to be a *typedef-name*"

In 10.1.7.3 [dcl.type.elab] paragraph 2, change "If the identifier resolves to a *typedef-name* or a template *type-parameter*" to "If the identifier resolves to a *typedef-name*".

This has been consolidated with the edits for some other issues. See N1376=02-0034.

---

516. Use of `signed` in bit-field declarations

Section: 10.1.7.2 [dcl.type.simple] **Status:** CD1 **Submitter:** comp.std.c++ **Date:** 25 Apr 2005

[Voted into WP at the October, 2006 meeting.]

10.1.7.2 [dcl.type.simple] paragraph 3 reads,

> It is implementation-defined whether bit-fields and objects of `char` type are represented as signed or unsigned quantities. The `signed` specifier forces `char` objects and bit-fields to be signed; it is redundant with other integral types.

The last sentence in that quote is misleading w.r.t. bit-fields. The first sentence in that quote is correct but incomplete.

Proposed fix: change the two sentences to read:

> It is implementation-defined whether objects of `char` type are represented as signed or unsigned quantities. The `signed` specifier forces `char` objects signed; it is redundant with other integral types except when declaring bit-fields (12.2.4 [class.bit]).

Proposed resolution (October, 2005):

Change 10.1.7.2 [dcl.type.simple] paragraph 3 as indicated:

> When multiple *simple-type-specifier*s are allowed, they can be freely intermixed with other *decl-specifier*s in any order. **[*Note:*** It is implementation-defined whether ~~bit-fields and~~ objects of `char` type **and certain bit-fields (12.2.4 [class.bit])** are represented as signed or unsigned quantities. The `signed` specifier forces **bit-fields and** `char` objects ~~and bit-fields~~ to be signed; it is redundant ~~with other integral types~~ **in other contexts**. ***—end note*****]**

---

651. Problems in `decltype` specification and examples

Section: 10.1.7.2 [dcl.type.simple] **Status:** CD1 **Submitter:** Daveed Vandevoorde **Date:** 16 Aug 2007

[Voted into the WP at the September, 2008 meeting.]

The second bullet of 10.1.7.2 [dcl.type.simple] paragraph 4 reads,

- otherwise, if `e` is a function call (8.2.2 [expr.call]) or an invocation of an overloaded operator (parentheses around `e` are ignored), `decltype(e)` is the return type of that function;

The reference to "that function" is imprecise; it is not the actual function called at runtime but the statically chosen function (ignoring covariant return types in virtual functions).

Also, the examples in this paragraph have errors:

1. The declaration of `struct A` should end with a semicolon.
2. The lines of the form `decltype(...);` are ill-formed; they need a declarator.

Proposed Resolution (October, 2007):

Change 10.1.7.2 [dcl.type.simple] paragraph 4 as follows:

> The type denoted by `decltype(e)` is defined as follows:
>
> - if `e` is an *id-expression* or a class member access (8.2.5 [expr.ref]), `decltype(e)` is the type of the entity named by `e`. If there is no such entity, or if `e` names a set of overloaded functions, the program is ill-formed;
> - otherwise, if `e` is a function call (8.2.2 [expr.call]) or an invocation of an overloaded operator (parentheses around `e` are ignored), `decltype(e)` is the return type of ~~that~~ **the statically chosen** function;
> - otherwise, if `e` is an lvalue, `decltype(e)` is `T&`, where `T` is the type of `e`;
> - otherwise, `decltype(e)` is the type of `e`.
>
> The operand of the `decltype` specifier is an unevaluated operand (clause 8 [expr]).
>
> [*Example:*
>
> ```
> const int&& foo();
> int i;
> struct A { double x; };
> const A* a = new A();
> decltype(foo()) x1;      // type is const int&&
> decltype(i) x2;          // type is int
> decltype(a->x) x3;       // type is double
> decltype((a->x)) x4;     // type is const double&
> ```
>
> —*end example*]

---

629. `auto` parsing ambiguity

Section: 10.1.7.4 [dcl.spec.auto] **Status:** CD1 **Submitter:** Daveed Vandevoorde **Date:** 14 March 2007

[Voted into the WP at the February, 2008 meeting as paper J16/08-0056 = WG21 N2546.]

We've found an interesting parsing ambiguity with the new meaning of `auto`. Consider:

```
typedef int T;
void f() {
    auto T = 42;  // Valid or not?
}
```

The question here is whether T should be a type specifier or a storage class? 10.1.7.4 [dcl.spec.auto] paragraph 1 says,

> The `auto` *type-specifier* has two meanings depending on the context of its use. In a *decl-specifier-seq* that contains at least one *type-specifier* (in addition to `auto`) that is not a *cv-qualifier*, the `auto` *type-specifier* specifies that the object named in the declaration has automatic storage duration.

In this case, `T` is a *type-specifier*, so the declaration is ill-formed: there is no *declarator-id*. Many, however, would like to see `auto` work "just like `int`," i.e., forcing `T` to be redeclared in the inner scope. Concerns cited included hijacking of the name in templates and inline function bodies over the course of time if a program revision introduces a type with that name in the surrounding context. Although it was pointed out that enclosing the name in parentheses in the inner declaration would prevent any such problems, this was viewed as unacceptably ugly.

Notes from the April, 2007 meeting:

The CWG wanted to avoid a rule like, "if `auto` can be a *type-specifier*, it is" (similar to the existing "if it can be a declaration, it is" rule) because of the lookahead and backtracking difficulties such an approach would pose for certain kinds of parsing techniques. It was noted

that the difficult lookahead cases all involve parentheses, which would not be a problem if only the "=" form of initializer were permitted in `auto` declarations; only very limited lookahead is required in that case. It was also pointed out that the "if it can be a *type-specifier*, it is" approach results in a quiet change of meaning for cases like

```
typedef int T;
int n = 3;
void f() {
    auto T(n);
}
```

This currently declares `n` to be an `int` variable in the inner scope but would, under the full lookahead approach, declare `T` to be a variable, quitely changing uses of `n` inside `f()` to refer to the outer variable.

The consensus of the CWG was to pursue the change to require the "=" form of initializer for `auto`.

Notes from the July, 2007 meeting:

See paper J16/07-0197 = WG21 N2337. There was no consensus among the CWG for either of the approaches recommended in the paper; additional input and direction is required.

---

172. Unsigned int as underlying type of enum

Section: 10.2 [dcl.enum] **Status:** CD1 **Submitter:** Bjarne Stroustrup **Date:** 26 Sep 1999

[Moved to DR at October 2002 meeting.]

According to 10.2 [dcl.enum] paragraph 5, the underlying type of an enum is an unspecified integral type, which could potentially be `unsigned int`. The promotion rules in 7.6 [conv.prom] paragraph 2 say that such an enumeration value used in an expression will be promoted to `unsigned int`. This means that a conforming implementation could give the value `false` for the following code:

```
enum { zero };
-1 < zero;      // might be false
```

This is counterintuitive. Perhaps the description of the underlying type of an enumeration should say that an unsigned underlying type can be used only if the values of the enumerators cannot be represented in the corresponding signed type. This approach would be consistent with the treatment of integral promotion of bitfields (7.6 [conv.prom] paragraph 3).

On a related note, 10.2 [dcl.enum] paragraph 5 says,

> the underlying type shall not be larger than `int` unless the value of an enumerator cannot fit in an `int` or `unsigned int`.

This specification does not allow for an enumeration like

```
enum { a = -1, b = UINT_MAX };
```

Since each enumerator can fit in an `int` or `unsigned int`, the underlying type is required to be no larger than `int`, even though there is no such type that can represent all the enumerators.

Proposed resolution (04/01; obsolete, see below):

Change 10.2 [dcl.enum] paragraph 5 as follows:

> It is implementation-defined which integral type is used as the underlying type for an enumeration except that the underlying type shall not be larger than `int` unless ~~the value of an enumerator cannot fit in an `int` or `unsigned int`~~ **neither `int` nor `unsigned int` can represent all the enumerator values. Furthermore, the underlying type shall not be an unsigned type if the corresponding signed type can represent all the enumerator values.**

See also issue 58.

Notes from 04/01 meeting:

It was noted that 7.6 [conv.prom] promotes unsigned types smaller than `int` to (signed) `int`. The signedness chosen by an implementation for small underlying types is therefore unobservable, so the last sentence of the proposed resolution above should apply only to `int` and larger types. This observation also prompted discussion of an alternative approach to resolving the issue, in which the b_{min} and b_{max} of the enumeration would determine the promoted type rather than the underlying type.

Proposed resolution (10/01):

Change 7.6 [conv.prom] paragraph 2 from

> An rvalue of type `wchar_t` (6.9.1 [basic.fundamental]) or an enumeration type (10.2 [dcl.enum]) 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`.

to

> An rvalue of type `wchar_t` (6.9.1 [basic.fundamental]) 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`. An rvalue of an enumeration type (10.2 [dcl.enum]) can be converted to an rvalue of the first of the following types that can represent all the values of the enumeration (i.e., the values in the range b_{min} to b_{max} as described in 10.2 [dcl.enum]): `int`, `unsigned int`, `long`, or `unsigned long`.

---

377. Enum whose enumerators will not fit in any integral type

Section: 10.2 [dcl.enum] **Status:** CD1 **Submitter:** Mark Mitchell **Date:** 30 August 2002

[Voted into WP at April 2003 meeting.]

10.2 [dcl.enum] defines the underlying type of an enumeration as an integral type "that can represent all the enumerator values defined in the enumeration".

What does the standard say about this code:

```
enum E { a = LONG_MIN, b = ULONG_MAX };
```

?

I think this should be ill-formed.

Proposed resolution:

In 10.2 [dcl.enum] paragraph 5 after

> The *underlying type* of an enumeration is an integral type that can represent all the enumerator values defined in the enumeration.

insert

> If no integral type can represent all the enumerator values, the enumeration is ill-formed.

---

518. Trailing comma following *enumerator-list*

Section: 10.2 [dcl.enum] **Status:** CD1 **Submitter:** Charles Bryant **Date:** 10 May 2005

[Voted into WP at April, 2006 meeting.]

The C language (since C99), and some C++ compilers, accept:

```
enum { FOO, };
```

as syntactically valid. It would be useful

- for machine generated code
- for minimising changes when editing
- to allow a distinction between the final item being intended as an ordinary item or as a limit:

```
enum { red, green, blue, num_colours };  // note no comma
enum { fred, jim, sheila, };             // last is not special
```

This proposed change is to permit a trailing comma in enum by adding:

enum *identifier*$_{opt}$ { *enumerator-list* , }

as an alternative definition for the *enum-specifier* nonterminal in 10.2 [dcl.enum] paragraph 1.

Proposed resolution (October, 2005):

Change the grammar in 10.2 [dcl.enum] paragraph 1 as indicated:

> *enum-specifier:*
>
> enum *identifier*$_{opt}$ { *enumerator-list*$_{opt}$ }
>
> enum ***identifier***$_{opt}$ { ***enumerator-list*** , }

---

660. Unnamed scoped enumerations

Section: 10.2 [dcl.enum] **Status:** CD1 **Submitter:** Daveed Vandevoorde **Date:** 15 November 2007

[Voted into the WP at the September, 2008 meeting.]

The current specification of scoped enumerations does not appear to forbid an example like the following, even though the enumerator `e` cannot be used:

```
enum class { e };
```

This might be covered by 10 [dcl.dcl] paragraph 3,

> In a *simple-declaration*, the optional *init-declarator-list* can be omitted only when declaring a class (clause 12 [class]) or enumeration (10.2 [dcl.enum]), that is, when the *decl-specifier-seq* contains either a *class-specifier*, an *elaborated-type-specifier* with a *class-key* (12.1 [class.name]), or an *enum-specifier*. In these cases and whenever a *class-specifier* or *enum-specifier* is present in the *decl-specifier-seq*, the identifiers in these specifiers are among the names being declared by the declaration (as *class-name*s, *enum-name*s, or *enumerator*s, depending on the syntax). In such cases, and except for the declaration of an unnamed bit-field (12.2.4 [class.bit]), the *decl-specifier-seq* shall introduce one or more names into the program, or shall redeclare a name introduced by a previous declaration.

which, when combined with paragraph 2,

> A declaration occurs in a scope (6.3 [basic.scope]); the scope rules are summarized in 6.4 [basic.lookup]. A declaration that declares a function or defines a class, namespace, template, or function also has one or more scopes nested within it. These nested scopes, in turn, can have declarations nested within them. Unless otherwise stated, utterances in clause 10 [dcl.dcl] about components in, of, or contained by a declaration or subcomponent thereof refer only to those components of the declaration that are *not* nested within scopes nested within the declaration.

appears to rule out the similar class definition,

```
struct { int m; };
```

However, a scoped enumeration is not listed in paragraph 2 among the constructs containing a nested scope (although 6.3.8 [basic.scope.enum] does describe "enumeration scope"); furthermore, an *enumerator-definition* is not formally a "nested declaration." If unusable scoped enumeration definitions are to be banned, these shortcomings in 10 [dcl.dcl] paragraph 2 must be addressed. (A note in 10.2 [dcl.enum] mentioning that unnamed scoped enumerations are not allowed would also be helpful.)

Notes from the February, 2008 meeting:

The consensus was to require that the *identifier* be present in an *enum-specifier* unless the *enum-key* is `enum`.

Proposed resolution (June, 2008):

Change 10.2 [dcl.enum] paragraph 2 as follows:

> ...The *enum-key*s `enum class` and `enum struct` are semantically equivalent; an enumeration type declared with one of these is a *scoped enumeration*, and its *enumerator*s are *scoped enumerators*. **The optional *identifier* shall not be omitted in the declaration of a scoped enumeration.** The *type-specifier-seq* of an *enum-base*...

---

540. Propagation of cv-qualifiers in reference-to-reference collapse

Section: 10.3.1 [namespace.def] **Status:** CD1 **Submitter:** Russell Yanofsky **Date:** 24 September 2005

[Voted into the WP at the October, 2006 meeting as part of paper J16/06-0188 = WG21 N2118.]

The resolution of issue 106 specifies that an attempt to create a type "reference to *cv1* `T`," where `T` is a typedef or template parameter of the type "reference to *cv2* `S`," actually creates the type "reference to *cv12* `S`," where *cv12* is the union of the two sets of cv-qualifiers.

One objection that has been raised to this resolution is that it is inconsistent with the treatment of cv-qualification and references specified in 11.3.2 [dcl.ref] paragraph 1, which says that cv-qualifiers applied to a typedef or template argument that is a reference type are ignored. For example:

```
typedef int& intref;
const intref r1;       // reference to int
const intref& r2;      // reference to const int
```

In fact, however, these two declarations are quite different. In the declaration of `r1`, `const` applies to a "top-level" reference, while in the declaration of `t2`, it occurs under a reference. In general, cv-qualifiers that appear under a reference are preserved, even if the type appears in a context in which top-level cv-qualification is removed, for example, in determining the type of a function from parameter types (11.3.5 [dcl.fct] paragraph 3) and in template argument deduction (17.9.2.1 [temp.deduct.call] paragraph 2).

Another objection to the resolution is that type composition gives different results in a single declaration than it does when separated into two declarations. For example:

```
template <class T>
struct X {
   typedef T const T_const;
   typedef T_const& type1;
   typedef T const& type2;
};

X<int&>::type1 t1;    // int&
X<int&>::type2 t2;    // int const&
```

The initial motivation for the propagation of cv-qualification during reference-to-reference collapse was to prevent inadvertent loss of cv-qualifiers in contexts in which it could make a difference. For example, if the resolution were changed to discard, rather than propagate, embedded cv-qualification, overload resolution could surprisingly select a non-const version of a member function:

```
struct X {
    void g();
    void g() const;
};

template <typename T> struct S {
    static void f(const T& t) {
        t.g();     // const or non-const???
    }
};

X x;

void q() {
    S<X>::f(x);    // calls X::g() const
    S<X&>::f(x);   // calls X::g()
}
```

Another potentially-surprising outcome of dropping embedded cv-qualifiers would be:

```
template <typename T> struct A {
    void f(T&);          // mutating version
    void f(const T&);    // non-mutating version
};

A<int&> ai;    // Ill-formed: A<int&> declares f(int&) twice
```

On the other hand, those who would like to see the resolution changed to discard embedded cv-qualifiers observe that these examples are too simple to be representative of real-world code. In general, it is unrealistic to expect that a template written with non-reference type parameters in mind will automatically work correctly with reference type parameters as a result of applying the issue 106 resolution. Instead, template metaprogramming allows the template author to choose explicitly whether cv-qualifiers are propagated or dropped, according to the intended use of the template, and it is more important to respect the reasonable intuition that a declaration involving a template parameter will not change the type that the parameter represents.

As a sample of real-world code, `trl::tuple` was examined. In both cases — the current resolution of issue 106 and one in which embedded cv-qualifiers were dropped — some metaprogramming was required to implement the intended interface, although the version reflecting the revised resolution was somewhat simpler.

Notes from the October, 2005 meeting:

The consensus of the CWG was that the resolution of issue 106 should be revised not to propagate embedded cv-qualification.

Note (February, 2006):

The wording included in the rvalue-reference paper, J16/06-0022 = WG21 N1952, incorporates changes intended to implement the October, 2005 consensus of the CWG.

---

11. How do the keywords typename/template interact with using-declarations?

Section: 10.3.3 [namespace.udecl] **Status:** CD1 **Submitter:** Bill Gibbons **Date:** unknown

[Voted into WP at March 2004 meeting.]

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 that they be used at every reference. The way to avoid `typename` at every use is to declare a typedef; then the typedef name itself is known to be a type. For *using-declaration*s, we decided that they do not introduce new declarations but rather are 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-declaration*s, 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-declaration*s (just as *class-key*s are disallowed now). Allow `typename` and `template` before unqualified names which refer to dependent qualified names through *using-declaration*s.
3. Document that this is broken.

Suggested Resolution:

The core WG already resolved this issue according to (1), but the wording does not seem to have been added to the standard. New wording needs to be drafted.

Issue 2:

Either way, one more point needs clarification. If the first option is adopted:

```
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 (10.3.3 [namespace.udecl] paragraph 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.) If the second option is adopted:

```
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.)

From John Spicer:

> 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.

Tentative Resolution:

The rules for multiple declarations with the same name in the same scope should treat a *using-declaration* which names a type as a typedef, just as a typedef of a class name is treated as a class declaration. This needs drafting work. Also see Core issue 36.

Rationale (04/99): Any semantics associated with the `typename` keyword in *using-declaration*s should be considered an extension.

Notes from the April 2003 meeting:

This was reopened because we are now considering extensions again. We agreed that it is desirable for the typename to be "sticky" on a using-declaration, i.e., references to the name introduced by the using-declaration are known to be type names without the use of the typename keyword (which can't be specified on an unqualified name anyway, as of now). The related issue with the template keyword already has a separate issue 109.

Issue 2 deals with the "struct hack." There is an example in 10.3.3 [namespace.udecl] paragraph 10 that shows a use of using-declarations to import two names that coexist because of the "struct hack." After some deliberation, we decided that the template-dependent using-declaration case is different enough that we did not have to support the "struct hack" in that case. A name introduced in such a case is like a typedef, and no other hidden type can be accessed through an elaborated type specifier.

Proposed resolution (April 2003, revised October 2003):

Add a new paragraph to the bottom of 10.3.3 [namespace.udecl]:

> If a *using-declaration* uses the keyword `typename` and specifies a dependent name (17.7.2 [temp.dep]), the name introduced by the *using-declaration* is treated as a *typedef-name* (10.1.3 [dcl.typedef]).

---

258. *using-declaration*s and cv-qualifiers

Section: 10.3.3 [namespace.udecl] **Status:** CD1 **Submitter:** Liam Fitzpatrick **Date:** 2 Nov 2000

[Voted into WP at April 2003 meeting.]

According to 10.3.3 [namespace.udecl] paragraph 12,

> When a *using-declaration* brings names from a base class into a derived class scope, member functions in the derived class override and/or hide member functions with the same name and parameter types in a base class (rather than conflicting).

Note that this description says nothing about the cv-qualification of the hiding and hidden member functions. This means, for instance, that a non-const member function in the derived class hides a const member function with the same name and parameter types instead of overloading it in the derived class scope. For example,

```
struct A {
  virtual int f() const;
  virtual int f();
};
struct B: A {
  B();
  int f();
  using A::f;
};

const B cb;
int i = cb.f(); // ill-formed: A::f() const hidden in B
```

The same terminology is used in 13.3 [class.virtual] paragraph 2:

> If a virtual member function `vf` is declared in a class `Base` and in a class `Derived`, derived directly or indirectly from `Base`, a member function `vf` with the same name and same parameter list as `Base::vf` is declared, then `Derived::vf` is also virtual (whether or not it is so declared) and it overrides `Base::vf`.

Notes on the 04/01 meeting:

The hiding and overriding should be on the basis of the function signature, which includes any cv-qualification on the function.

Proposed resolution (04/02):

In 10.3.3 [namespace.udecl] paragraph 12 change:

> When a *using-declaration* brings names from a base class into a derived class scope, member functions in the derived class override and/or hide member functions with the same name and parameter types in a base class (rather than conflicting).

to read:

> When a *using-declaration* brings names from a base class into a derived class scope, member functions and member function templates in the derived class override and/or hide member functions and member function templates with the same name, parameter-type-list (11.3.5 [dcl.fct]), and *cv*-qualification in a base class (rather than conflicting).

In 13.3 [class.virtual] paragraph 2 change:

> If a virtual member function `vf` is declared in a class `Base` and in a class `Derived`, derived directly or indirectly from `Base`, a member function `vf` with the same name and same parameter list as `Base::vf` is declared, then `Derived::vf` is also virtual (whether or not it is so declared) and it overrides `Base::vf`.

to read:

> If a virtual member function `vf` is declared in a class `Base` and in a class `Derived`, derived directly or indirectly from `Base`, a member function `vf` with the same name, parameter-type-list (11.3.5 [dcl.fct]), and *cv*-qualification as `Base::vf` is declared, then `Derived::vf` is also virtual (whether or not it is so declared) and it overrides `Base::vf`.

See issue 140 for the definition of *parameter-type-list*.

---

460. Can a *using-declaration* name a namespace?

Section: 10.3.3 [namespace.udecl] **Status:** CD1 **Submitter:** John Spicer **Date:** 12 Feb 2004

[Voted into WP at April 2005 meeting.]

Can a using-declaration be used to import a namespace?

```
namespace my_namespace{
  namespace my_namespace2 {
    int function_of_my_name_space(){ return 2;}
  }
}

int main (){
  using  ::my_namespace::my_namespace2;
  return my_namespace2::function_of_my_name_space();
}
```

Several popular compilers give an error on this, but there doesn't seem to be anything in 10.3.3 [namespace.udecl] that prohibits it. It should be noted that the user can get the same effect by using a namespace alias:

```
namespace my_namespace2 = ::my_namespace::my_namespace2;
```

Notes from the March 2004 meeting:

We agree that it should be an error.

Proposed resolution (October, 2004):

Add the following as a new paragraph after 10.3.3 [namespace.udecl] paragraph 5:

> A *using-declaration* shall not name a namespace;

---

4. Does extern "C" affect the linkage of function names with internal linkage?

Section: 10.5 [dcl.link] **Status:** CD1 **Submitter:** Mike Anderson **Date:** unknown

[Moved to DR at 4/01 meeting.]

10.5 [dcl.link] paragraph 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 10.5 [dcl.link] paragraph 6 doesn't apply.

Suggested 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.

Proposed Resolution:

The intent is to distingush *implicit* linkage from explicit linkage for both name linkage and language (function type) linkage. (It might be more clear to use the terms *name linkage* and *type linkage* to distinguish these concepts. A function can have a name with one kind of linkage and a type with a different kind of linkage. The function itself has no linkage: it has no name, only the declaration has a name. This becomes more obvious when you consider function pointers.)

The tentatively agreed proposal is to apply implicit linkage to names declared in brace-enclosed linkage specifications and to non-top-level names declared in simple linkage specifications; and to apply explicit linkage to top-level names declared in simple linkage specifications.

The language linkage of any function type formed through a function declarator is that of the nearest enclosing *linkage-specification*. For purposes of determining whether the declaration of a namespace-scope name matches a previous declaration, the language linkage portion of the type of a function declaration (that is, the language linkage of the function itself, not its parameters, return type or exception specification) is ignored.

For a *linkage-specification* using braces, i.e.

> `extern` *string-literal* `{` *declaration-seq*$_{\text{opt}}$ `}`

the linkage of any declaration of a namespace-scope name (including local externs) which is not contained in a nested *linkage-specification*, is not declared to have no linkage (static), and does not match a previous declaration is given the linkage specified in the *string-literal.* The language linkage of the type of any function declaration of a namespace-scope name (including local externs) which is not contained in a nested *linkage-specification* and which is declared with function declarator syntax is the same as that of a matching previous declaration, if any, else is specified by *string-literal*.

For a *linkage-specification* without braces, i.e.

> `extern` *string-literal declaration*

the linkage of the names declared in the top-level declarators of *declaration* is specified by *string-literal*, if this conflicts with the linkage of any matching previous declarations, the program is ill-formed. The language linkage of the type of any top-level function declarator is specified by *string-literal*, if this conflicts with the language linkage of the type of any matching previous function declarations, the program is ill-formed. The effect of the *linkage-specification* on other (non top-level) names declared in *declaration* is the same as that of the brace-enclosed form.

Bill Gibbons: In particular, these should be well-formed:

```
extern "C" void f(void (*fp)());   // parameter type is pointer to
                                   // function with C language linkage
extern "C++" void g(void (*fp)()); // parameter type is pointer to
                                   // function with C++ language linkage

extern "C++" {                     // well-formed: the linkage of "f"
    void f(void(*fp)());           // and the function type used in the
}                                  // parameter still "C"

extern "C" {                       // well-formed: the linkage of "g"
    void g(void(*fp)());           // and the function type used in the
}                                  // parameter still "C++"
```

but these should not:

```
extern "C++" void f(void(*fp)());  // error - linkage of "f" does not
                                   // match previous declaration
                                   // (linkage of function type used in
                                   // parameter is still "C" and is not
                                   // by itself ill-formed)
extern "C" void g(void(*fp)());    // error - linkage of "g" does not
                                   // match previous declaration
                                   // (linkage of function type used in
                                   // parameter is still "C++" and is not
                                   // by itself ill-formed)
```

That is, non-top-level declarators get their linkage from matching declarations, if any, else from the nearest enclosing linkage specification. (As already described, top-level declarators in a brace-enclosed linkage specification get the linkage from matching declarations, if any, else from the linkage specifcation; while top-level declarators in direct linkage specifications get their linkage from that specification.)

Mike Miller: This is a pretty significant change from the current specification, which treats the two forms of language linkage similarly for most purposes. I don't understand why it's desirable to expand the differences.

It seems very unintuitive to me that you could have a top-level declaration in an `extern "C"` block that would *not* receive "C" linkage.

In the current standard, the statement in 10.5 [dcl.link] paragraph 4 that

> the specified language linkage applies to the function types of all function declarators, function names, and variable names introduced by the declaration(s)

applies to both forms. I would thus expect that in

```
extern "C" void f(void(*)());
extern "C++" {
    void f(void(*)());
}
extern "C++" f(void(*)());
```

both "C++" declarations would be well-formed, declaring an overloaded version of `f` that takes a pointer to a "C++" function as a parameter. I wouldn't expect that either declaration would be a redeclaration (valid or invalid) of the "C" version of `f`.

Bill Gibbons: The potential difficulty is the matching process and the handling of deliberate overloading based on language linkage. In the above examples, how are these two declarations matched:

```
extern "C" void f(void (*fp1)());

extern "C++" {
    void f(void(*fp2)());
}
```

given that the linkage that is part of fp1 is "C" while the linkage (prior to the matching process) that is part of fp2 is "C++"?

The proposal is that the linkage which is part of the parameter type is not determined until after the match is attempted. This almost always correct because you can't overload "C" and "C++" functions; so if the function names match, it is likely that the declarations are supposed to be the same.

Mike Miller: This seems like more trouble than it's worth. This comparison of function types ignoring linkage specifications is, as far as I know, not found anywhere in the current standard. Why do we need to invent it?

Bill Gibbons: It is possible to construct pathological cases where this fails, e.g.

```
extern "C" typedef void (*PFC)();  // pointer to "C" linkage function
void f(PFC);         // parameter is pointer to "C" function
void f(void (*)());  // matching declaration or overload based on
                     // difference in linkage type?
```

It is reasonable to require explicit typedefs in this case so that in the above example the second function declaration gets its parameter type function linkage from the first function declaration.

(In fact, I think you can't get into this situation without having already used typedefs to declare different language linkage for the top-level and parameter linkages.)

For example, if the intent is to overload based on linkage a typedef is needed:

```
extern "C" typedef void (*PFC)();  // pointer to "C" linkage function
void f(PFC);              // parameter is pointer to "C" function
typedef void (*PFCPP)();  // pointer to "C++" linkage function
void f(PFCPP);            // parameter is pointer to "C++" function
```

In this case the two function declarations refer to different functions.

Mike Miller: This seems pretty strange to me. I think it would be simpler to determine the type of the parameter based on the containing linkage specification (implicitly "C++") and require a typedef if the user wants to override the default behavior. For example:

```
extern "C" {
    typedef void (*PFC)();    // pointer to "C" function
    void f(void(*)());        // takes pointer to "C" function
}

void f(void(*)());            // new overload of "f", taking
                              // pointer to "C++" function

void f(PFC);                  // redeclare extern "C" version
```

Notes from 04/00 meeting:

The following changes were tentatively approved, but because they do not completely implement the proposal above the issue is being kept for the moment in "drafting" status.

Notes from 10/00 meeting:

After further discussion, the core language working group determined that the more extensive proposal described above is not needed and that the following changes are sufficient.

Proposed resolution (04/01):

1. Change the first sentence of 10.5 [dcl.link] paragraph 1 from

   > All function types, function names, and variable names have a language linkage.

   to

   > All function types, function names with external linkage, and variable names with external linkage have a language linkage.

2. Change the following sentence of 10.5 [dcl.link] paragraph 4:

   > In a *linkage-specification*, the specified language linkage applies to the function types of all function declarators, function names, and variable names introduced by the declaration(s).

   to

   > In a *linkage-specification*, the specified language linkage applies to the function types of all function declarators, function names with external linkage, and variable names with external linkage declared within the *linkage-specification*.

3. Add at the end of the final example on 10.5 [dcl.link] paragraph 4:

```
extern "C" {
  static void f4();    // the name of the function f4 has
                       // internal linkage (not C language
                       // linkage) and the function's type
                       // has C language linkage
}
extern "C" void f5() {
  extern void f4();    // Okay -- name linkage (internal)
                       // and function type linkage (C
                       // language linkage) gotten from
                       // previous declaration.
}
extern void f4();      // Okay -- name linkage (internal)
                       // and function type linkage (C
                       // language linkage) gotten from
                       // previous declaration.
void f6() {
  extern void f4();    // Okay -- name linkage (internal)
                       // and function type linkage (C
                       // language linkage) gotten from
                       // previous declaration.
}
```

4. Change 10.5 [dcl.link] paragraph 7 from

   > Except for functions with internal linkage, a function first declared in a *linkage-specification* behaves as a function with external linkage. [*Example:*
   >
   > ```
   > extern "C" double f();
   > static double f();     // error
   > ```
   >
   > is ill-formed (10.1.1 [dcl.stc]).] The form of *linkage-specification* that contains a braced-enclosed *declaration-seq* does not affect whether the contained declarations are definitions or not (6.1 [basic.def]); the form of *linkage-specification* directly containing a single declaration is treated as an `extern` specifier (10.1.1 [dcl.stc]) for the purpose of determining whether the contained declaration is a definition. [*Example:*
   >
   > ```
   > extern "C" int i;       // declaration
   > extern "C" {
   >       int i;           // definition
   > }
   > ```
   >
   > —*end example*] A *linkage-specification* directly containing a single declaration shall not specify a storage class. [*Example:*
   >
   > ```
   > extern "C" static void f(); // error
   > ```

—end example]

to

A declaration directly contained in a *linkage-specification* is treated as if it contains the `extern` specifier (10.1.1 [dcl.stc]) for the purpose of determining the linkage of the declared name and whether it is a definition. Such a declaration shall not specify a storage class. [*Example:*

```
extern "C" double f();
static double f();     // error
extern "C" int i;      // declaration
extern "C" {
        int i;         // definition
}
extern "C" static void g(); // error
```

—end example]

---

29. Linkage of locally declared functions

Section: 10.5 [dcl.link] **Status:** CD1 **Submitter:** Mike Ball **Date:** 19 Mar 1998

[Moved to DR at October 2002 meeting. This was incorrectly marked as having DR status between 4/01 and 4/02. It was overlooked when issue 4 was moved to DR at the 4/01 meeting; this one should have been moved as well, because it's resolved by the changes there.]

Consider the following:

```
extern "C" void foo()
{
    extern void bar();
    bar();
}
```

Does "bar()" have "C" language linkage?

The ARM is explicit and says

> A linkage-specification for a function also applies to functions and objects declared within it.

The DIS says

> In a *linkage-specification*, the specified language linkage applies to the function types of all function declarators, function names, and variable names introduced by the declaration(s).

Is the body of a function definition part of the declaration?

From Mike Miller:

Yes: from 10 [dcl.dcl] paragraph 1,

> *declaration:*
> *function-definition*

and 11.4 [dcl.fct.def] paragraph 1:

> *function-definition:*
> *decl-specifier-seq*$_{opt}$ *declarator ctor-initializer*$_{opt}$ *function-body*

At least that's how I'd read it.

From Dag Brück:

Consider the following where extern "C" has been moved to a separate declaration:

```
extern "C" void foo();

void foo() { extern void bar(); bar(); }
```

I think the ARM wording could possibly be interpreted such that bar() has "C" linkage in my example, but not the DIS wording.

As a side note, I have always wanted to think that placing extern "C" on a function definition or a separate declaration would produce identical programs.

Proposed Resolution (04/01):

See the proposed resolution for Core issue 4, which covers this case.

The ODR should also be checked to see whether it addresses name and type linkage.

---

686. Type declarations/definitions in *type-specifier-seq*s and *type-id*s

Section: 11.1 [dcl.name] **Status:** CD1 **Submitter:** Jens Maurer **Date:** 21 March, 2008

[Voted into the WP at the September, 2008 meeting.]

The restrictions on declaring and/or defining classes inside *type-specifier-seq*s and *type-id*s are inconsistent throughout the Standard. This is probably due to the fact that nearly all of the sections that deal with them attempt to state the restriction afresh. There are three cases:

1. 8.3.4 [expr.new], 9.4 [stmt.select], and 15.3.2 [class.conv.fct] prohibit "declarations" of classes and enumerations. That means that

   ```
   while (struct C* p = 0) ;
   ```

   is ill-formed unless a prior declaration of `C` has been seen. These appear to be cases that should have been fixed by issue 379, changing "class declaration" to "class definition," but were overlooked.

2. 8.1.5 [expr.prim.lambda], 10 [dcl.dcl], and 11.3.5 [dcl.fct] (late-specified return types) do not contain any restriction at all.

3. All the remaining cases prohibit "type definitions," apparently referring to classes and enumerations.

Suggested resolution:

Add something like, "A class or enumeration shall not be defined in a *type-specifier-seq* or in a *type-id*," to a single place in the Standard and remove all other mentions of that restriction (allowing declarations via *elaborated-type-specifier*).

Mike Miller:

An *alias-declaration* is just a different syntax for a typedef declaration, which allows definitions of a class in the type; I would expect the same to be true of an *alias-declaration*. I don't have any particularly strong attachment to allowing a class definition in an *alias-declaration*. My only concern is introducing an irregularity into what are currently exact-match semantics with typedefs.

There's a parallel restriction in many (but not all?) of these places on typedef declarations.

Jens Maurer:

Those are redundant, as `typedef` is not a *type-specifier*, and should be removed as well.

Proposed resolution (March, 2008):

1. Delete the indicated words from 8.2.7 [expr.dynamic.cast] paragraph 1:

   ...~~Types shall not be defined in a `dynamic_cast`:~~...

2. Delete the indicated words from 8.2.8 [expr.typeid] paragraph 4:

   ...~~Types shall not be defined in the *type-id*.~~...

3. Delete the indicated words from 8.2.9 [expr.static.cast] paragraph 1:

   ...~~Types shall not be defined in a `static_cast`:~~...

4. Delete the indicated words from 8.2.10 [expr.reinterpret.cast] paragraph 1:

   ...~~Types shall not be defined in a `reinterpret_cast`:~~...

5. Delete the indicated words from 8.2.11 [expr.const.cast] paragraph 1:

   ...~~Types shall not be defined in a `const_cast`:~~...

6. Delete paragraph 5 of 8.3.3 [expr.sizeof]:

   ~~Types shall not be defined in a `sizeof` expression.~~

7. Delete paragraph 5 of 8.3.4 [expr.new]:

   ~~The *type-specifier-seq* shall not contain class declarations, or enumeration declarations.~~

8. Delete paragraph 4 of 8.3.6 [expr.alignof]:

   ~~A type shall not be defined in an `alignof` expression.~~

9. Delete paragraph 3 of 8.4 [expr.cast]:

   ~~Types shall not be defined in casts.~~

10. Delete the indicated words from 9.4 [stmt.select] paragraph 2:

    ...~~The *type-specifier-seq* shall not contain `typedef` and shall not declare a new class or enumeration.~~...

11. Add the indicated words to 10.1.7 [dcl.type] paragraph 3:

At least one *type-specifier* that is not a *cv-qualifier* is required in a declaration unless it declares a constructor, destructor or conversion function. [*Footnote:* ...] **A *type-specifier-seq* shall not define a class or enumeration unless it appears in the *type-id* of an *alias-declaration* (10.1.3 [dcl.typedef]).**

12. Delete the indicated words from 15.3.2 [class.conv.fct] paragraph 1:

    ...~~Classes, enumerations, and *typedef-names* shall not be declared in the *type-specifier-seq*~~....

13. Delete the indicated words from 18.3 [except.handle] paragraph 1:

    ...~~Types shall not be defined in an *exception-declaration*.~~

14. Delete paragraph 6 of 18.4 [except.spec]:

    ~~Types shall not be defined in *exception-specifications*.~~

[Drafting note: no changes are required to 8.1.5 [expr.prim.lambda], 10.1.3 [dcl.typedef], 10.6.2 [dcl.align], 10.2 [dcl.enum], 11.3.5 [dcl.fct], 17.1 [temp.param], or 17.2 [temp.names].]

---

160. Missing `std::` qualification

Section: 11.2 [dcl.ambig.res] **Status:** CD1 **Submitter:** Al Stevens **Date:** 23 Aug 1999

[Moved to DR at 10/01 meeting.]

11.2 [dcl.ambig.res] paragraph 3 shows an example that includes `<cstddef>` with no using declarations or directives and refers to `size_t` without the `std::` qualification.

Many references to `size_t` throughout the document omit the `std::` namespace qualification.

This is a typical case. The use of `std::` is inconsistent throughout the document.

In addition, the use of exception specifications should be examined for consistency.

(See also issue 282.)

Proposed resolution:

In 4.6 [intro.execution] paragraph 9, replace all two instances of "`sig_atomic_t`" by "`std::sig_atomic_t`".

In 6.1 [basic.def] paragraph 4, replace all three instances of "`string`" by "`std::string`" in the example and insert "`#include <string>`" at the beginning of the example code.

In 6.6.1 [basic.start.main] paragraph 4, replace

> Calling the function
>
> ```
> void exit(int);
> ```
>
> declared in `<cstdlib>`...

by

> Calling the function `std::exit(int)` declared in `<cstdlib>`...

and also replace "`exit`" by "`std::exit`" in the last sentence of that paragraph.

In 6.6.1 [basic.start.main] first sentence of paragraph 5, replace "`exit`" by "`std::exit`".

In 6.6.2 [basic.start.static] paragraph 4, replace "`terminate`" by "`std::terminate`".

In 6.6.3 [basic.start.dynamic] paragraph 1, replace "`exit`" by "`std::exit`" (see also issue 28).

In 6.6.3 [basic.start.dynamic] paragraph 3, replace all three instances of "`atexit`" by "`std::atexit`" and both instances of "`exit`" by "`std::exit`" (see also issue 28).

In 6.6.3 [basic.start.dynamic] paragraph 4, replace

> Calling the function
>
> ```
> void abort();
> ```
>
> declared in `<cstdlib>`...

by

> Calling the function `std::abort()` declared in `<cstdlib>`...

and "`atexit`" by "`std::atexit`" (see also issue 28).

In 6.7.4.1 [basic.stc.dynamic.allocation] paragraph 1 third sentence, replace "`size_t`" by "`std::size_t`".

In 6.7.4.1 [basic.stc.dynamic.allocation] paragraph 3, replace "`new_handler`" by "`std::new_handler`". Furthermore, replace "`set_new_handler`" by "`std::set_new_handler`" in the note.

In 6.7.4.1 [basic.stc.dynamic.allocation] paragraph 4, replace "`type_info`" by "`std::type_info`" in the note.

In 6.7.4.2 [basic.stc.dynamic.deallocation] paragraph 3, replace all four instances of "`size_t`" by "`std::size_t`".

In 6.8 [basic.life] paragraph 5, replace "`malloc`" by "`std::malloc`" in the example code and insert "`#include <cstdlib>`" at the beginning of the example code.

In 6.9 [basic.types] paragraph 2, replace "`memcpy`" by "`std::memcpy`" (the only instance in the footnote and both instances in the example) and replace "`memmove`" by "`std::memmove`" in the footnote (see also issue 43).

In 6.9 [basic.types] paragraph 3, replace "`memcpy`" by "`std::memcpy`", once in the normative text and once in the example (see also issue 43).

In 6.9.1 [basic.fundamental] paragraph 8 last sentence, replace "`numeric_limits`" by "`std::numeric_limits`".

In 8.2.7 [expr.dynamic.cast] paragraph 9 second sentence, replace "`bad_cast`" by "`std::bad_cast`".

In 8.2.8 [expr.typeid] paragraph 2, replace "`type_info`" by "`std::type_info`" and "`bad_typeid`" by "`std::bad_typeid`".

In 8.2.8 [expr.typeid] paragraph 3, replace "`type_info`" by "`std::type_info`".

In 8.2.8 [expr.typeid] paragraph 4, replace both instances of "`type_info`" by "`std::type_info`".

In 8.3.3 [expr.sizeof] paragraph 6, replace both instances of "`size_t`" by "`std::size_t`".

In 8.3.4 [expr.new] paragraph 11 last sentence, replace "`size_t`" by "`std::size_t`".

In 8.7 [expr.add] paragraph 6, replace both instances of "`ptrdiff_t`" by "`std::ptrdiff_t`".

In 8.7 [expr.add] paragraph 8, replace "`ptrdiff_t`" by "`std::ptrdiff_t`".

In 9.6 [stmt.jump] paragraph 2, replace "`exit`" by "`std::exit`" and "`abort`" by "`std::abort`" in the note.

In 11.2 [dcl.ambig.res] paragraph 3, replace "`size_t`" by "`std::size_t`" in the example.

In 11.4 [dcl.fct.def] paragraph 5, replace "`printf`" by "`std::printf`" in the note.

In 15.4 [class.dtor] paragraph 13, replace "`size_t`" by "`std::size_t`" in the example.

In 15.5 [class.free] paragraph 2, replace all four instances of "`size_t`" by "`std::size_t`" in the example.

In 15.5 [class.free] paragraph 6, replace both instances of "`size_t`" by "`std::size_t`" in the example.

In 15.5 [class.free] paragraph 7, replace all four instances of "`size_t`" by "`std::size_t`" in the two examples.

In 15.7 [class.cdtor] paragraph 4, replace "`type_info`" by "`std::type_info`".

In 16.6 [over.built] paragraph 13, replace all five instances of "`ptrdiff_t`" by "`std::ptrdiff_t`".

In 16.6 [over.built] paragraph 14, replace "`ptrdiff_t`" by "`std::ptrdiff_t`".

In 16.6 [over.built] paragraph 21, replace both instances of "`ptrdiff_t`" by "`std::ptrdiff_t`".

In 17.2 [temp.names] paragraph 4, replace both instances of "`size_t`" by "`std::size_t`" in the example. (The example is quoted in issue 96.)

In 17.3 [temp.arg] paragraph 1, replace "`complex`" by "`std::complex`", once in the example code and once in the comment.

In 17.8.3 [temp.expl.spec] paragraph 8, issue 24 has already corrected the example.

In 18.1 [except.throw] paragraph 6, replace "`uncaught_exception`" by "`std::uncaught_exception`".

In 18.1 [except.throw] paragraph 7, replace "`terminate`" by "`std::terminate`" and both instances of "`unexpected`" by "`std::unexpected`".

In 18.1 [except.throw] paragraph 8, replace "`terminate`" by "`std::terminate`".

In 18.2 [except.ctor] paragraph 3, replace "`terminate`" by "`std::terminate`".

In 18.3 [except.handle] paragraph 9, replace "`terminate`" by "`std::terminate`".

In 18.4 [except.spec] paragraph 8, replace "`unexpected`" by "`std::unexpected`".

In 18.4 [except.spec] paragraph 9, replace "`unexpected`" by "`std::unexpected`" and "`terminate`" by "`std::terminate`".

In 18.5 [except.special] paragraph 1, replace "`terminate`" by "`std::terminate`" and "`unexpected`" by "`std::unexpected`".

In the heading of 18.5.1 [except.terminate], replace "`terminate`" by "`std::terminate`".

In 18.5.1 [except.terminate] paragraph 1, footnote in the first bullet, replace "`terminate`" by "`std::terminate`". In the same paragraph, fifth bullet, replace "`atexit`" by "`std::atexit`". In the same paragraph, last bullet, replace "`unexpected_handler`" by "`std::unexpected_handler`".

In 18.5.1 [except.terminate] paragraph 2, replace

> In such cases,
>
> ```
> void terminate();
> ```
>
> is called...

by

> In such cases, `std::terminate()` is called...

and replace all three instances of "`terminate`" by "`std::terminate`".

In the heading of _N4606_.15.5.2 [except.unexpected], replace "`unexpected`" by "`std::unexpected`".

In _N4606_.15.5.2 [except.unexpected] paragraph 1, replace

> ...the function
>
> ```
> void unexpected();
> ```
>
> is called...

by

> ...the function `std::unexpected()` is called...

.

In _N4606_.15.5.2 [except.unexpected] paragraph 2, replace "`unexpected`" by "`std::unexpected`" and "`terminate`" by "`std::terminate`".

In _N4606_.15.5.2 [except.unexpected] paragraph 3, replace "`unexpected`" by "`std::unexpected`".

In the heading of 18.5.2 [except.uncaught], replace "`uncaught_exception`" by "`std::uncaught_exception`".

In 18.5.2 [except.uncaught] paragraph 1, replace

> The function
>
> ```
> bool uncaught_exception()
> ```
>
> returns `true`...

by

> The function `std::uncaught_exception()` returns `true`...

.

In the last sentence of the same paragraph, replace "`uncaught_exception`" by "`std::uncaught_exception`".

---

112. Array types and cv-qualifiers

Section: 11.3.4 [dcl.array] **Status:** CD1 **Submitter:** Steve Clamage **Date:** 4 May 1999

[Moved to DR at 10/01 meeting.]

Steve Clamage: Section 11.3.4 [dcl.array] paragraph 1 reads in part as follows:

> Any type of the form "*cv-qualifier-seq* array of `N T`" is adjusted to "array of `N` *cv-qualifier-seq* `T`," and similarly for "array of unknown bound of `T`." [*Example:*
>
> ```
> typedef int A[5], AA[2][3];
> typedef const A CA;     // type is "array of 5 const int"
> typedef const AA CAA;   // type is "array of 2 array of 3 const int"
> ```
>
> —*end example*] [*Note:* an "array of `N` *cv-qualifier-seq* `T`" has cv-qualified type; such an array has internal linkage unless explicitly declared `extern` (10.1.7.1 [dcl.type.cv]) and must be initialized as specified in 11.6 [dcl.init] .]

The Note appears to contradict the sentence that precedes it.

Mike Miller: I disagree; all it says is that whether the qualification on the element type is direct ("`const int x[5]`") or indirect ("`const A CA`"), the array itself is qualified in the same way the elements are.

Steve Clamage: In addition, section 6.9.3 [basic.type.qualifier] paragraph 2 says:

> A compound type (6.9.2 [basic.compound]) is not cv-qualified by the cv-qualifiers (if any) of the types from which it is compounded. Any cv-qualifiers applied to an array type affect the array element type, not the array type (11.3.4 [dcl.array])."

The Note appears to contradict that section as well.

Mike Miller: Yes, but consider the last two sentences of 6.9.3 [basic.type.qualifier] paragraph 5:

> Cv-qualifiers applied to an array type attach to the underlying element type, so the notation "*cv* `T`," where `T` is an array type, refers to an array whose elements are so-qualified. Such array types can be said to be more (or less) cv-qualified than other types based on the cv-qualification of the underlying element types.

I think this says essentially the same thing as 11.3.4 [dcl.array] paragraph 1 and its note: the qualification of an array is (bidirectionally) equivalent to the qualification of its members.

Mike Ball: I find this a very far reach. The text in 11.3.4 [dcl.array] is essentially that which is in the C standard (and is a change from early versions of C++). I don't see any justification at all for the bidirectional equivalence. It seems to me that the note is left over from the earlier version of the language.

Steve Clamage: Finally, the Note seems to say that the declaration

```
volatile char greet[6] = "Hello";
```

gives "greet" internal linkage, which makes no sense.

Have I missed something, or should that Note be entirely removed?

Mike Miller: At least the wording in the note should be repaired not to indicate that volatile-qualification gives an array internal linkage. Also, depending on how the discussion goes, either the wording in 6.9.3 [basic.type.qualifier] paragraph 2 or in paragraph 5 needs to be amended to be consistent regarding whether an array type is considered qualified by the qualification of its element type.

Steve Adamczyk pointed out that the current state of affairs resulted from the need to handle reference binding consistently. The wording is intended to define the question, "Is an array type cv-qualified?" as being equivalent to the question, "Is the element type of the array cv-qualified?"

Proposed resolution (10/00):

Replace the portion of the note in 11.3.4 [dcl.array] paragraph 1 reading

> such an array has internal linkage unless explicitly declared `extern` (10.1.7.1 [dcl.type.cv]) and must be initialized as specified in 11.6 [dcl.init].

with

> see 6.9.3 [basic.type.qualifier].

---

140. Agreement of parameter declarations

Section: 11.3.5 [dcl.fct] **Status:** CD1 **Submitter:** Steve Clamage **Date:** 15 Jul 1999

[Moved to DR at 10/01 meeting.]

11.3.5 [dcl.fct] paragraph 3 says,

> All declarations for a function with a given parameter list shall agree exactly both in the type of the value returned and in the number and type of parameters.

It is not clear what this requirement means with respect to a pair of declarations like the following:

```
int f(const int);
int f(int x) { ... }
```

Do they violate this requirement? Is `x const` in the body of the function declaration?

Tom Plum: I think the FDIS quotation means that the pair of decls are valid. But it doesn't clearly answer whether `x` is `const` inside the function definition. As to intent, I *know* the intent was that if the function definition wants to specify that `x` is `const`, the `const` must appear specifically in the defining decl, not just on some decl elsewhere. But I can't prove that intent from the drafted words.

Mike Miller: I think the intent was something along the following lines:

> Two function declarations denote the same entity if the names are the same and the function signatures are the same. (Two function declarations with C language linkage denote the same entity if the names are the same.) All declarations of a given function shall agree exactly both in the type of the value returned and in the number and type of parameters; the presence or absence of the ellipsis is considered part of the signature.

(See 6.5 [basic.link] paragraph 9. That paragraph talks about names in different scopes and says that function references are the same if the "types are identical for purposes of overloading," i.e., the signatures are the same. See also 10.5 [dcl.link] paragraph 6 regarding C language linkage, where only the name is required to be the same for declarations in different namespaces to denote the same function.)

According to this paragraph, the type of a parameter is determined by considering its *decl-specifier-seq* and *declarator* and then applying the array-to-pointer and function-to-pointer adjustments. The *cv-qualifier* and storage class adjustments are performed for the function type but not for the parameter types.

If my interpretation of the intent of the second sentence of the paragraph is correct, the two declarations in the example violate that restriction — the parameter types are not the same, even though the function types are. Since there's no dispensation mentioned for "no diagnostic required," an implementation presumably must issue a diagnostic in this case. (I think "no diagnostic required" should be stated if the declarations occur in different translation units — unless there's a blanket statement to that effect that I have forgotten?)

(I'd also note in passing that, if my interpretation is correct,

```
void f(int);
void f(register int) { }
```

is also an invalid pair of declarations.)

Proposed resolution (10/00):

1. In 3 [intro.defs] "**signature**," change "the types of its parameters" to "its parameter-type-list (11.3.5 [dcl.fct])".
2. In the third bullet of 6.5 [basic.link] paragraph 9 change "the function types are identical for the purposes of overloading" to "the parameter-type-lists of the functions (11.3.5 [dcl.fct]) are identical."
3. In the sub-bullets of the third bullet of 8.2.5 [expr.ref] paragraph 4, change all four occurrences of "function of (parameter type list)" to "function of parameter-type-list."
4. In 11.3.5 [dcl.fct] paragraph 3, change

   > All declarations for a function with a given parameter list shall agree exactly both in the type of the value returned and in the number and type of parameters; the presence or absence of the ellipsis is considered part of the function type.

   to

   > All declarations for a function shall agree exactly in both the return type and the parameter-type-list.

5. In 11.3.5 [dcl.fct] paragraph 3, change

   > The resulting list of transformed parameter types is the function's *parameter type list*.

   to

   > The resulting list of transformed parameter types and the presence or absence of the ellipsis is the function's *parameter-type-list*.

6. In 11.3.5 [dcl.fct] paragraph 4, change "the parameter type list" to "the parameter-type-list."
7. In the second bullet of 16.1 [over.load] paragraph 2, change all occurrences of "parameter types" to "parameter-type-list."
8. In 16.3 [over.match] paragraph 1, change "the types of the parameters" to "the parameter-type-list."
9. In the last sub-bullet of the third bullet of 16.3.1.2 [over.match.oper] paragraph 3, change "parameter type list" to "parameter-type-list."

Note, 7 Sep 2001:

Editorial changes while putting in issue 147 brought up the fact that injected-class-name is not a syntax term and therefore perhaps shouldn't be written with hyphens. The same can be said of parameter-type-list.

---

262. Default arguments and ellipsis

Section: 11.3.5 [dcl.fct] **Status:** CD1 **Submitter:** Jamie Schmeiser **Date:** 13 Nov 2000

[Voted into WP at April 2003 meeting.]

The interaction of default arguments and ellipsis is not clearly spelled out in the current wording of the Standard. 11.3.6 [dcl.fct.default] paragraph 4 says,

> In a given function declaration, all parameters subsequent to a parameter with a default argument shall have default arguments supplied in this or previous declarations.

Strictly speaking, ellipsis isn't a parameter, but this could be clearer. Also, in 11.3.5 [dcl.fct] paragraph 2,

> If the *parameter-declaration-clause* terminates with an ellipsis, the number of arguments shall be equal to or greater than the number of parameters specified.

This could be interpreted to refer to the number of arguments after the addition of default arguments to the argument list given in the call expression, but again it could be clearer.

Notes from 04/01 meeting:

The consensus opinion was that an ellipsis is not a parameter and that default arguments should be permitted preceding an ellipsis.

Proposed Resolution (4/02):

Change the following sentence in 11.3.5 [dcl.fct] paragraph 2 from

> If the *parameter-declaration-clause* terminates with an ellipsis, the number of arguments shall be equal to or greater than the number of parameters specified.

to

> If the *parameter-declaration-clause* terminates with an ellipsis, the number of arguments shall be equal to or greater than the number of parameters that do not have a default argument.

As noted in the defect, section 11.3.6 [dcl.fct.default] is correct but could be clearer.

In 11.3.6 [dcl.fct.default], add the following as the first line of the example in paragraph 4.

```
void g(int = 0, ...);  // okay, ellipsis is not a parameter so it can follow
                       // a parameter with a default argument
```

---

295. cv-qualifiers on function types

Section: 11.3.5 [dcl.fct] **Status:** CD1 **Submitter:** Nathan Sidwell **Date:** 29 Jun 2001

[Moved to DR at October 2002 meeting.]

This concerns the inconsistent treatment of cv qualifiers on reference types and function types. The problem originated with GCC bug report c++/2810. The bug report is available at http://gcc.gnu.org/cgi-bin/gnatsweb.pl?cmd=view&pr=2810&database=gcc

11.3.2 [dcl.ref] describes references. Of interest is the statement (my emphasis)

> Cv-qualified references are ill-formed **except** when the cv-qualifiers are introduced through the use of a typedef or of a template type argument, in which case the cv-qualifiers are ignored.

Though it is strange to ignore 'volatile' here, that is not the point of this defect report. 11.3.5 [dcl.fct] describes function types. Paragraph 4 states,

> In fact, if at any time in the determination of a type a cv-qualified function type is formed, the program is ill-formed.

No allowance for typedefs or template type parameters is made here, which is inconsistent with the equivalent reference case.

The GCC bug report was template code which attempted to do,

```
template <typename T> void foo (T const &);
void baz ();
...
foo (baz);
```

in the instantiation of foo, `T` is `` `void ()' `` and an attempt is made to const qualify that, which is ill-formed. This is a surprise.

Suggested resolution:

Replace the quoted sentence from paragraph 4 in 11.3.5 [dcl.fct] with

> cv-qualified functions are ill-formed, except when the cv-qualifiers are introduced through the use of a typedef or of a template type argument, in which case the cv-qualifiers are ignored.

Adjust the example following to reflect this.

Proposed resolution (10/01):

In 11.3.5 [dcl.fct] paragraph 4, replace

> The effect of a *cv-qualifier-seq* in a function declarator is not the same as adding cv-qualification on top of the function type, i.e., it does not create a cv-qualified function type. In fact, if at any time in the determination of a type a cv-qualified function type is formed, the program is ill-formed. [*Example:*
>
> ```
> typedef void F();
> struct S {
>   const F f;          // ill-formed
> };
> ```
>
> *-- end example*]

by

> The effect of a *cv-qualifier-seq* in a function declarator is not the same as adding cv-qualification on top of the function type. In the latter case, the cv-qualifiers are ignored. [*Example:*
>
> ```
> typedef void F();
> struct S {
>   const F f;          // ok; equivalent to void f();
> };
> ```
>
> *-- end example*]

Strike the last bulleted item in 17.9.2 [temp.deduct] paragraph 2, which reads

> Attempting to create a cv-qualified function type.

Nathan Sidwell comments (18 Dec 2001): The proposed resolution simply states attempts to add cv qualification on top of a function type are ignored. There is no mention of whether the function type was introduced via a typedef or template type parameter. This would appear to allow

```
void (const *fptr) ();
```

but, that is not permitted by the grammar. This is inconsistent with the wording of adding cv qualifiers to a reference type, which does mention typedefs and template parameters, even though

```
int &const ref;
```

is also not allowed by the grammar.

Is this difference intentional? It seems needlessly confusing.

Notes from 4/02 meeting:

Yes, the difference is intentional. There is no way to add cv-qualifiers other than those cases.

Notes from April 2003 meeting:

Nathan Sidwell pointed out that some libraries use the inability to add const to a type T as a way of testing that T is a function type. He will get back to us if he has a proposal for a change.

---

681. Restrictions on declarators with late-specified return types

Section: 11.3.5 [dcl.fct] **Status:** CD1 **Submitter:** Mike Miller **Date:** 10 March, 2008

[Voted into the WP at the September, 2008 meeting as part of paper N2757.]

The wording added to 11.3.5 [dcl.fct] for declarators with late-specified return types says,

> In a declaration `T D` where `D` has the form
>
> > `D1 (` *parameter-declaration-clause* `)` *cv-qualifier-seq*$_{opt}$ *ref-qualifier*$_{opt}$ *exception-specification*$_{opt}$ `->` *type-id*
>
> and the type of the contained *declarator-id* in the declaration `T D1` is "*derived-declarator-type-list* `T`," `T` shall be the single *type-specifier* `auto` and the *derived-declarator-type-list* shall be empty.

These restrictions were intended to ensure that the return type of the function is exactly the specified *type-id* following the `->`, not modified by declarator operators and cv-qualification.

Unfortunately, the requirement for an empty *derived-declarator-type-list* does not achieve this goal but instead forbids declarations like

```
auto (*fp)() -> int;      // pointer to function returning int
```

while allowing declarations like

```
auto *f() -> int;         // function returning pointer to int
```

The reason for this is that, according to the grammar in 11 [dcl.decl] paragraph 4, the declarator `*f() -> int` is parsed as a *ptr-operator* applied to the *direct-declarator* `f() -> int`; that is, the declarator `D1` seen in 11.3.5 [dcl.fct] is just `f`, and the *derived-declarator-type-list* is thus empty.

By contrast, the declarator `(*fp)() -> int` is parsed as the *direct-declarator* `(*fp)` followed by the *parameter-declaration-clause*, etc. In this case, `D1` in 11.3.5 [dcl.fct] is `(*fp)` and the *derived-declarator-type-list* is "pointer to," i.e., not empty.

My personal view is that there is no reason to forbid the `(*fp)() -> int` form, and that doing so is problematic. For example, this restriction would require users desiring the late-specified return type syntax to write function parameters as function types and rely on parameter type transformations rather than writing them as pointer-to-function types, as they will actually turn out to be:

```
void f(auto (*fp)() -> int);  // ill-formed
void f(auto fp() -> int);     // OK (but icky)
```

It may be helpful in deciding whether to allow this form to consider the example of a function returning a pointer to a function. With the current restriction, only one of the three plausible forms is allowed:

```
auto (*f())() -> int;           // Disallowed
auto f() -> int (*)();          // Allowed
auto f() -> auto (*)() -> int;  // Disallowed
```

Suggested resolution:

1. Delete the words "and the *derived-declarator-type-list* shall be empty" from 11.3.5 [dcl.fct] paragraph 2.
2. Add a new paragraph following 11 [dcl.decl] paragraph 4:

A *ptr-operator* shall not be applied, directly or indirectly, to a function declarator with a late-specified return type (11.3.5 [dcl.fct]).

Proposed resolution (June, 2008):

1. Change the grammar in 11 [dcl.decl] paragraph 4 as follows:

declarator:
~~*direct-declarator*~~
~~*ptr-operator declarator*~~
ptr-declarator
noptr-declarator parameters-and-qualifiers -> ***type-id***

ptr-declarator:
noptr-declarator
ptr-operator ptr-declarator

~~*direct-declarator:*~~
noptr-declarator:
declarator-id
~~*direct-declarator* (*parameter-declaration-clause*)~~
~~*cv-qualifier-seq*$_{opt}$ *ref-qualifier*$_{opt}$ *exception-specification*$_{opt}$~~
~~*direct-declarator* (*parameter-declaration-clause*)~~
~~*cv-qualifier-seq*$_{opt}$ *ref-qualifier*$_{opt}$ *exception-specification*$_{opt}$ -> *type-id*~~
~~*direct-declarator* [*constant-expression*$_{opt}$]~~
noptr-declarator parameters-and-qualifiers
noptr-declarator [***constant-expression***$_{opt}$]
(***ptr-****declarator*)

parameters-and-qualifiers:
(***parameter-declaration-clause***) ***cv-qualifier-seq***$_{opt}$ ***ref-qualifier***$_{opt}$ ***exception-specification***$_{opt}$
...

2. Change the grammar in 11.1 [dcl.name] paragraph 1 as follows:

...
abstract-declarator:
~~*ptr-operator abstract-declarator*$_{opt}$~~
~~*direct-abstract-declarator*~~
ptr-abstract-declarator
noptr-abstract-declarator$_{opt}$ ***parameters-and-qualifiers*** -> ***type-id***
. . .

ptr-abstract-declarator:
noptr-abstract-declarator
ptr-operator ptr-abstract-declarator$_{opt}$

~~*direct-abstract-declarator:*~~
~~*direct-abstract-declarator*$_{opt}$ (*parameter-declaration-clause*)~~
~~*cv-qualifier-seq*$_{opt}$ *ref-qualifier*$_{opt}$ *exception-specification*$_{opt}$~~
~~*direct-abstract-declarator*$_{opt}$ (*parameter-declaration-clause*)~~
~~*cv-qualifier-seq*$_{opt}$ *ref-qualifier*$_{opt}$ *exception-specification*$_{opt}$ -> *type-id*~~
~~*direct-abstract-declarator*$_{opt}$ [*constant-expression*$_{opt}$]~~
noptr-abstract-declarator:
noptr-abstract-declarator$_{opt}$ ***parameters-and-qualifiers***
noptr-abstract-declarator$_{opt}$ [***constant-expression***$_{opt}$]
(***ptr-****abstract-declarator*)

3. Change 11.3.5 [dcl.fct] paragraph 2 as follows:

... `T` shall be the single *type-specifier* `auto` ~~and the *derived-declarator-type-list* shall be empty~~. Then the type...

4. Change all occurrences of *direct-new-declarator* in 8.3.4 [expr.new] to *noptr-new-declarator*. These changes appear in the grammar in paragraph 1 and in the text of paragraphs 6-8, as follows:

...
new-declarator:
ptr-operator new-declarator$_{opt}$
~~*direct-*~~***noptr-****new-declarator*

~~*direct-*~~***noptr-****new-declarator:*
[*expression*]
~~*direct-*~~***noptr-****new-declarator* [*constant-expression*]
...

When the allocated object is an array (that is, the ~~*direct-*~~***noptr-****new-declarator* syntax is used or the *new-type-id* or *type-id* denotes an array type), the *new-expression* yields a pointer to the initial element (if any) of the array. [*Note:* both `new int` and `new int[10]` have type `int*` and the type of `new int[i][10]` is `int (*)[10]` —*end note*]

Every *constant-expression* in a ~~*direct-*~~***noptr-****new-declarator* shall be an integral constant expression (8.20 [expr.const]) and evaluate to a strictly positive value. The *expression* in a ~~*direct-*~~***noptr-****new-declarator* shall be of integral type, enumeration type, or a class type for which a single non-explicit conversion function to integral or enumeration type exists (15.3 [class.conv]). If the expression is of class type, the expression is converted by calling that conversion function, and the result of the conversion is used in place of the original expression. If the value of the expression is negative, the behavior is undefined. [*Example:* given the definition `int n = 42`, `new float[n][5]` is well-formed (because `n` is the *expression* of a ~~*direct-*~~***noptr-****new-declarator*), but `new float[5][n]` is ill-formed (because `n` is not a constant expression). If `n` is negative, the effect of `new float[n][5]` is undefined. —*end example*]

When the value of the *expression* in a ~~*direct-*~~***noptr-****new-declarator* is zero, the allocation function is called to allocate an array with no elements.

---

136. Default arguments and friend declarations

Section: 11.3.6 [dcl.fct.default] **Status:** CD1 **Submitter:** Daveed Vandevoorde **Date:** 9 July 1999

[Moved to DR at 10/01 meeting.]

11.3.6 [dcl.fct.default] paragraph 4 says,

> For non-template functions, default arguments can be added in later declarations of a function in the same scope. Declarations in different scopes have completely distinct sets of default arguments. That is, declarations in inner scopes do not acquire default arguments from declarations in outer scopes, and vice versa.

It is unclear how this wording applies to friend function declarations. For example,

```
void f(int, int, int=0);             // #1
class C {
    friend void f(int, int=0, int);  // #2
};
void f(int=0, int, int);             // #3
```

Does the declaration at #2 acquire the default argument from #1, and does the one at #3 acquire the default arguments from #2?

There are several related questions involved with this issue:

1. Is the friend declaration in the scope of class C or in the surrounding namespace scope?

   Mike Miller: 11.3.6 [dcl.fct.default] paragraph 4 is speaking about the lexical location of the declaration... The friend declaration occurs in a different declarative region from the declaration at #1, so I would read [this paragraph] as saying that it starts out with a clean slate of default arguments.

   Bill Gibbons: Yes. It occurs in a different region, although it declares a name in the same region (i.e. a redeclaration). This is the same as with local externs and is intended to work the same way. We decided that local extern declarations cannot add (beyond the enclosing block) new default arguments, and the same should apply to friend declarations.

   John Spicer: The question is whether [this paragraph] does (or should) mean declarations that appear in the same lexical scope or declarations that declare names in the same scope. In my opinion, it really needs to be the latter. It seems somewhat paradoxical to say that a friend declaration declares a function in namespace scope yet the declaration in the class still has its own attributes. To make that work I think you'd have to make friends more like block externs that really do introduce a name into the scope in which the declaration is contained.

2. Should default arguments be permitted in friend function declarations, and what effect should they have?

   Bill Gibbons: In the absence of a declaration visible in class scope to which they could be attached, default arguments on friend declarations do not make sense. [They should be] ill-formed, to prevent surprises.

   John Spicer: It is important that the following case work correctly:

   ```
   class X {
           friend void f(X x, int i = 1){}
   };

   int main()
   {
           X x;
           f(x);
   }
   ```

   In other words, a function first declared in a friend declaration must be permitted to have default arguments and those default arguments must be usable when the function is found by argument dependent lookup. The reason that this is important is that it is common practice to *define* functions in friend declarations in templates, and that definition is the only place where the default arguments can be specified.

3. What restrictions should be placed on default argument usage with friend declarations?

John Spicer: We want to avoid instantiation side effects. IMO, the way to do this would be to prohibit a friend declaration from providing default arguments if a declaration of that function is already visible. Once a function has had a default specified in a friend declaration it should not be possible to add defaults in another declaration be it a friend or normal declaration.

Mike Miller: The position that seems most reasonable to me is to allow default arguments in friend declarations to be used in Koenig lookup, but to say that they are completely unrelated to default arguments in declarations in the surrounding scope; and to forbid use of a default argument in a call if more than one declaration in the overload set has such a default, as in the proposed resolution for issue 1.

(See also issues 21, 95, 138, 139, 143, 165, and 166.)

Notes from 10/99 meeting:

Four possible outcomes were identified:

1. If a friend declaration declares a default parameter, allow no other declarations of that function in the translation unit.
2. Same as preceding, but only allow the friend declaration if it is also a definition.
3. Disallow default arguments in friend declarations.
4. Treat the default arguments in each friend declaration as a distinct set, causing an error if the call would be ambiguous.

The core group eliminated the first and fourth options from consideration, but split fairly evenly between the remaining two.

A straw poll of the full committee yielded the following results (given as number favoring/could live with/"over my dead body"):

1. 0/14/5
2. 8/13/5
3. 11/7/14
4. 7/10/9

Additional discussion is recorded in the "Record of Discussion" for the meeting, J16/99-0036 = WG21 N1212. See also paper J16/00-0040 = WG21 N1263.

Proposed resolution (10/00):

In 11.3.6 [dcl.fct.default], add following paragraph 4:

> If a friend declaration specifies a default argument expression, that declaration must be a definition and shall be the only declaration of the function or function template in the translation unit.

---

5. CV-qualifiers and type conversions

Section: 11.6 [dcl.init] **Status:** CD1 **Submitter:** Josee Lajoie **Date:** unknown

[Moved to DR at 4/01 meeting.]

The description of copy-initialization in 11.6 [dcl.init] paragraph 14 says:

- If the destination type is a (possibly cv-qualified) class type:
  ...
- Otherwise (i.e. for the remaining copy-initialization cases), user-defined conversion sequences that can convert from the source type to the destination type or (when a conversion function is used) to a derived class thereof are enumerated ... if the function is a constructor, the call initializes a temporary of the destination type. ...

Should "destination type" in this last bullet refer to "cv-unqualified destination type" to make it clear that the destination type excludes any cv-qualifiers? This would make it clearer that the following example is well-formed:

```
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.

Steve Adamczyk:

Because of late changes to this area, the relevant text is now the third sub-bullet of the fourth bullet of 11.6 [dcl.init] paragraph 14:

> Otherwise (i.e., for the remaining copy-initialization cases), user-defined conversion sequences that can convert from the source type to the destination type or (when a conversion function is used) to a derived class thereof are enumerated... The function

selected is called with the initializer expression as its argument; if the function is a constructor, the call initializes a temporary of the destination type. The result of the call (which is the temporary for the constructor case) is then used to direct-initialize, according to the rules above, the object that is the destination of the copy-initialization.

The issue still remains whether the wording should refer to "the cv-unqualified version of the destination type." I think it should.

Notes from 10/00 meeting:

The original example does not illustrate the remaining problem. The following example does:

```
struct C { };
C c;
struct A {
    A(const A&);
    A(const C&);
};
const volatile A a = c;    // Okay
```

Proposed Resolution (04/01):

In 11.6 [dcl.init], paragraph 14, bullet 4, sub-bullet 3, change

> if the function is a constructor, the call initializes a temporary of the destination type.

to

> if the function is a constructor, the call initializes a temporary of the cv-unqualified version of the destination type.

---

78. Section 8.5 paragraph 9 should state it only applies to non-static objects

Section: 11.6 [dcl.init] **Status:** CD1 **Submitter:** Judy Ward **Date:** 15 Dec 1998

Paragraph 9 of 11.6 [dcl.init] says:

> If no initializer is specified for an object, and the object is of (possibly cv-qualified) non-POD class type (or array thereof), the object shall be default-initialized; if the object is of const-qualified type, the underlying class type shall have a user-declared default constructor. Otherwise, if no initializer is specified for an object, the object and its subobjects, if any, have an indeterminate initial value; if the object or any of its subobjects are of const-qualified type, the program is ill-formed.

It should be made clear that this paragraph does not apply to static objects.

Proposed resolution (10/00): In 11.6 [dcl.init] paragraph 9, replace

> Otherwise, if no initializer is specified for an object..."

with

> Otherwise, if no initializer is specified for a **non-static** object...

---

177. Lvalues vs rvalues in copy-initialization

Section: 11.6 [dcl.init] **Status:** CD1 **Submitter:** Steve Adamczyk **Date:** 25 October 1999

[Moved to DR at 4/02 meeting.]

Is the temporary created during copy-initialization of a class object treated as an lvalue or an rvalue? That is, is the following example well-formed or not?

```
struct B { };
struct A {
    A(A&);    // not const
    A(const B&);
};
B b;
A a = b;
```

According to 11.6 [dcl.init] paragraph 14, the initialization of `a` is performed in two steps. First, a temporary of type `A` is created using `A::A(const B&)`. Second, the resulting temporary is used to direct-initialize `a` using `A::A(A&)`.

The second step requires binding a reference to non-const to the temporary resulting from the first step. However, 11.6.3 [dcl.init.ref] paragraph 5 requires that such a reference be bound only to lvalues.

It is not clear from 6.10 [basic.lval] whether the temporary created in the process of copy-initialization should be treated as an lvalue or an rvalue. If it is an lvalue, the example is well-formed, otherwise it is ill-formed.

Proposed resolution (04/01):

1. In 11.6 [dcl.init] paragraph 14, insert the following after "the call initializes a temporary of the destination type":

   The temporary is an rvalue.

2. In 18.1 [except.throw] paragraph 3, replace

   The temporary is used to initialize the variable...

   with

   The temporary is an lvalue and is used to initialize the variable...

(See also issue 84.)

---

277. Zero-initialization of pointers

Section: 11.6 [dcl.init] **Status:** CD1 **Submitter:** Andrew Sawyer **Date:** 5 Apr 2001

[Moved to DR at 10/01 meeting.]

The intent of 11.6 [dcl.init] paragraph 5 is that pointers that are zero-initialized will contain a null pointer value. Unfortunately, the wording used,

> ...set to the value of 0 (zero) converted to `T`

does not match the requirements for creating a null pointer value given in 7.11 [conv.ptr] paragraph 1:

> A *null pointer constant* is an integral constant expression (8.20 [expr.const]) rvalue of integer type that evaluates to zero. A null pointer constant can be converted to a pointer type; the result is the *null pointer value* of that type...

The problem is that the "value of 0" in the description of zero-initialization is not specified to be an integral constant expression. Nonconstant expressions can also have the value 0, and converting a nonconst 0 to pointer type need not result in a null pointer value.

Proposed resolution (04/01):

In 11.6 [dcl.init] paragraph 5, change

> ...set to the value 0 (zero) converted to `T`;

to

> ...set to the value 0 (zero), taken as an integral constant expression, converted to `T`; [*footnote:* as specified in 7.11 [conv.ptr], converting an integral constant expression whose value is 0 to a pointer type results in a null pointer value.]

---

302. Value-initialization and generation of default constructor

Section: 11.6 [dcl.init] **Status:** CD1 **Submitter:** Steve Adamczyk **Date:** 23 Jul 2001

[Moved to DR at October 2002 meeting.]

We've been looking at implementing value-initialization. At one point, some years back, I remember Bjarne saying that something like X() in an expression should produce an X object with the same value one would get if one created a static X object, i.e., the uninitialized members would be zero-initialized because the whole object is initialized at program startup, before the constructor is called.

The formulation for default-initialization that made it into TC1 (in 11.6 [dcl.init]) is written a little differently (see issue 178), but I had always assumed that it would still be a valid implementation to zero the whole object and then call the default constructor for the troublesome "non-POD but no user-written constructor" cases.

That almost works correctly, but I found a problem case:

```
struct A {
  A();
  ~A();
};
struct B {
  // B is a non-POD with no user-written constructor.
  // It has a nontrivial generated constructor.
  const int i;
  A a;
};
int main () {
  // Value-initializing a "B" doesn't call the default constructor for
  // "B"; it value-initializes the members of B.  Therefore it shouldn't
  // cause an error on generation of the default constructor for the
  // following:
  new B();
}
```

If the definition of the `B::B()` constructor is generated, an error is issued because the const member "`i`" is not initialized. But the definition of value-initialization doesn't require calling the constructor, and therefore it doesn't require generating it, and therefore the error shouldn't be detected.

So this is a case where zero-initializing and then calling the constructor is not equivalent to value-initializing, because one case generates an error and the other doesn't.

This is sort of unfortunate, because one doesn't want to generate all the required initializations at the point where the "`()`" initialization occurs. One would like those initializations to be packaged in a function, and the default constructor is pretty much the function one wants.

I see several implementation choices:

1. Zero the object, then call the default generated constructor. This is not valid unless the standard is changed to say that the default constructor might be generated for value-initialization cases like the above (that is, it's implementation-dependent whether the constructor definition is generated). The zeroing operation can of course be optimized, if necessary, to hit only the pieces of the object that would otherwise be left uninitialized. An alternative would be to *require* generation of the constructor for value-initialization cases, even if the implementation technique doesn't call the constructor at that point. It's pretty likely that the constructor is going to have to be generated at some point in the program anyway.
2. Make a new value-initialization "constructor," whose body looks a lot like the usual generated constructor, but which also zeroes other members. No errors would be generated while generating this modified constructor, because it generates code that does full initialization. (Actually, it wouldn't guarantee initialization of reference members, and that might be an argument for generating the constructor, in order to get that error.) This is standard-conforming, but it destroys object-code compatibility.
3. Variation on (1): Zero first, and generate the object code for the default constructor when it's needed for value-initialization cases, but don't issue any errors at that time. Issue the errors only if it turns out the constructor is "really" referenced. Aside from the essential shadiness of this approach, I fear that something in the generation of the constructor will cause a template instantiation which will be an abservable side effect.

Personally, I find option 1 the least objectionable.

Proposed resolution (10/01):

Add the indicated wording to the third-to-last sentence of 6.2 [basic.def.odr] pararaph 2:

> A default constructor for a class is used by default initialization **or value initialization** as specified in 11.6 [dcl.init].

Add a footnote to the indicated bullet in 11.6 [dcl.init] paragraph 5:

- if `T` is a non-union class type without a user-declared constructor, then every non-static data member and base-class component of `T` is value-initialized. **[Footnote: Value-initialization for such a class object may be implemented by zero-initializing the object and then calling the default constructor.]**

Add the indicated wording to the first sentence of 15.1 [class.ctor] paragraph 7:

> An implicitly-declared default constructor for a class is *implicitly defined* when it is used **(6.2 [basic.def.odr])** to create an object of its class type (4.5 [intro.object]).

---

509. Dead code in the specification of default initialization

Section: 11.6 [dcl.init] **Status:** CD1 **Submitter:** Mike Miller **Date:** 18 Mar 2005

[Voted into the WP at the September, 2008 meeting (resolution in paper N2762).]

The definition of default initialization (11.6 [dcl.init] paragraph 5) is:

- if `T` is a non-POD class type (clause 12 [class]), the default constructor for `T` is called (and the initialization is ill-formed if `T` has no accessible default constructor);
- if `T` is an array type, each element is default-initialized;
- otherwise, the object is zero-initialized.

However, default initialization is invoked only for non-POD class types and arrays thereof (8.3.4 [expr.new] paragraph 15 for *new-expression*s, 11.6 [dcl.init] paragraph 10 for top-level objects, and 15.6.2 [class.base.init] paragraph 4 for member and base class subobjects — but see issue 510). Consequently, all cases that invoke default initialization are handled by the first two bullets; the third bullet can never be reached. Its presence is misleading, so it should be removed.

Notes from the September, 2008 meeting:

The approach adopted in the resolution in paper N2762 was different from the suggestion above: it changes the definition of default initialization to include POD types and changes the third bullet to specify that "no initialization is performed."

---

543. Value initialization and default constructors

Section: 11.6 [dcl.init] **Status:** CD1 **Submitter:** Mike Miller **Date:** 27 October 2005

[Voted into the WP at the September, 2008 meeting (resolution in paper N2762).]

The wording resulting from the resolution of issue 302 does not quite implement the intent of the issue. The revised wording of 6.2 [basic.def.odr] paragraph 2 is:

> A default constructor for a class is used by default initialization or value initialization as specified in 11.6 [dcl.init].

This sounds as if 11.6 [dcl.init] specifies how and under what circumstances value initialization uses a default constructor (which was, in fact, the case for default initialization in the original wording). However, the normative text there makes it plain that value initialization does *not* call the default constructor (the permission granted to implementations to call the default constructor for value initialization is in a non-normative footnote).

The example that occasioned this observation raises an additional question. Consider:

```
struct POD {
  const int x;
};

POD data = POD();
```

According to the (revised) resolution of issue 302, this code is ill-formed because the implicitly-declared default constructor will be implicitly defined as a result of being used by value initialization (15.1 [class.ctor] paragraph 7), and the implicitly-defined constructor fails to initialize a const-qualified member (15.6.2 [class.base.init] paragraph 4). This seems unfortunate, because the (trivial) default constructor of a POD class is otherwise not used — default initialization applies only to non-PODs — and it is not actually needed in value initialization. Perhaps value initialization should be defined to "use" the default constructor only for non-POD classes? If so, both of these problems would be resolved by rewording the above-referenced sentence of 6.2 [basic.def.odr] paragraph 2 as:

> A default constructor for a **non-POD** class is used by default initialization or value initialization ~~as specified in~~ **(**11.6 [dcl.init]**)**.

Notes from the April, 2006 meeting:

The approach favored by the CWG was to leave 6.2 [basic.def.odr] unchanged and to add normative wording to 11.6 [dcl.init] indicating that it is unspecified whether the default constructor is called.

Notes from the October, 2006 meeting:

The CWG now prefers that it should not be left unspecified whether programs of this sort are well- or ill-formed; instead, the Standard should require that the default constructor be defined in such cases. Three possibilities of implementing this decision were discussed:

1. Change 6.2 [basic.def.odr] to state flatly that the default constructor is used by value initialization (removing the implication that 11.6 [dcl.init] determines the conditions under which it is used).
2. Change 11.6 [dcl.init] to specify that non-union class objects with no user-declared constructor are value-initialized by first zero-initializing the object and then calling the (implicitly-defined) default constructor, replacing the current specification of value-initializing each of its sub-objects.
3. Add a normative statement to 11.6 [dcl.init] that value-initialization causes the implicitly-declared default constructor to be implicitly defined, even if it is not called.

Proposed resolution (June, 2008):

Change the second bullet of the value-initialization definition in 11.6 [dcl.init] paragraph 5 as follows:

- if `T` is a non-union class type without a user-provided constructor, then ~~every non-static data member and base-class component of `T` is value-initialized; [*Footnote:* Value-initialization for such a class object may be implemented by zero-initializing the object and then calling the default constructor. — *end footnote*]~~ **the object is zero-initialized and the implicitly-defined default constructor is called;**

Notes from the September, 2008 meeting:

The resolution supplied in paper N2762 differs from the June, 2008 proposed resolution in that the implicitly-declared default constructor is only called (and thus defined) if it is non-trivial, making the `struct POD` example above well-formed.

---

430. Ordering of expression evaluation in initializer list

Section: 11.6.1 [dcl.init.aggr] **Status:** CD1 **Submitter:** Nathan Sidwell **Date:** 23 July 2003

[Voted into the WP at the April, 2007 meeting as part of paper J16/07-0099 = WG21 N2239.]

A recent GCC bug report (http://gcc.gnu.org/bugzilla/show_bug.cgi?id=11633) asks about the validity of

```
int count = 23;
int foo[] = { count++, count++, count++ };
```

is this undefined or unspecified or something else? I can find nothing in 11.6.1 [dcl.init.aggr] that indicates whether the components of an initializer-list are evaluated in order or not, or whether they have sequence points between them.

6.7.8/23 of the C99 std has this to say

> The order in which any side effects occur among the initialization list expressions is unspecified.

I think similar wording is needed in 11.6.1 [dcl.init.aggr]

Steve Adamczyk: I believe the standard is clear that each initializer expression in the above is a full-expression (4.6 [intro.execution]/12-13; see also issue 392) and therefore there is a sequence point after each expression (4.6 [intro.execution]/16). I agree that the standard does not seem to dictate the order in which the expressions are evaluated, and perhaps it should. Does anyone know of a compiler that would not evaluate the expressions left to right?

Mike Simons: Actually there is one, that does not do left to right: gcc/C++. None of the post increment operations take effect until after the statement finishes. So in the sample code gcc stores 23 into all positions in the array. The commercial vendor C++ compilers for AIX, Solaris, Tru64, HPUX (parisc and ia64), and Windows, all do sequence points at each ',' in the initializer list.

---

491. Initializers for empty-class aggregate members

Section: 11.6.1 [dcl.init.aggr] **Status:** CD1 **Submitter:** Nathan Sidwell **Date:** 15 Dec 2004

[Voted into WP at April, 2007 meeting.]

The current wording of 11.6.1 [dcl.init.aggr] paragraph 8 requires that

> An *initializer* for an aggregate member that is an empty class shall have the form of an empty initializer-list `{}`.

This is overly constraining. There is no reason that the following should be ill-formed:

```
struct S { };
S s;
S arr[1] = { s };
```

Mike Miller: The wording of 11.6.1 [dcl.init.aggr] paragraph 8 is unclear. "An aggregate member" would most naturally mean "a member of an aggregate." In context, however, I think it must mean "a member [of an aggregate] that is an aggregate", that is, a subaggregate. Members of aggregates need not themselves be aggregates (cf paragraph 13 and 15.6.1 [class.expl.init]); it cannot be the case that an object of an empty class with a user-declared constructor must be initialized with `{}` when it is a member of an aggregate. This wording should be clarified, regardless of the decision on Nathan's point.

Proposed resolution (October, 2005):

This issue is resolved by the resolution of issue 413.

---

632. Brace-enclosed initializer for scalar member of aggregate

Section: 11.6.1 [dcl.init.aggr] **Status:** CD1 **Submitter:** Greg Comeau **Date:** 3 May 2007

[Voted into the WP at the June, 2008 meeting as part of paper N2672.]

C (both C90 and C99) appear to allow a declaration of the form

```
struct S { int i; } s = { { 5 } };
```

in which the initializer of a scalar member of an aggregate can itself be brace-enclosed. The relevant wording from the C99 Standard is found in 6.7.8 paragraph 11:

> The initializer for a scalar shall be a single expression, optionally enclosed in braces.

and paragraph 16:

> Otherwise, the initializer for an object that has aggregate or union type shall be a brace-enclosed list of initializers for the elements or named members.

The "list of initializers" in paragraph 16 must be a recursive reference to paragraph 11 (that's the only place that describes how an initialized item gets its value from the initializer expression), which would thus make the "brace-enclosed" part of paragraph 11 apply to each of the initializers in the list in paragraph 16 as well.

This appears to be an incompatibility between C and C++: 11.6.1 [dcl.init.aggr] paragraph 11 says,

> If the *initializer-list* begins with a left brace, then the succeeding comma-separated list of *initializer-clause*s initializes the members of a subaggregate....

which clearly leaves the impression that only a subaggregate may be initialized by a brace-enclosed *initializer-clause*.

Either the specification in 11.6.1 [dcl.init.aggr] should be changed to allow a brace-enclosed initializer of a scalar member of an aggregate, as in C, or this incompatibility should be listed in Appendix C [diff].

Notes from the July, 2007 meeting:

It was noted that implementations differ in their handling of this construct; however, the issue is long-standing and fairly obscure.

Notes from the October, 2007 meeting:

The initializer-list proposal will resolve this issue when it is adopted.

---

291. Overload resolution needed when binding reference to class rvalue

Section: 11.6.3 [dcl.init.ref] **Status:** CD1 **Submitter:** Andrei Iltchenko **Date:** 15 Jun 2001

[Voted into WP at October 2005 meeting.]

There is a place in the Standard where overload resolution is implied but the way that a set of candidate functions is to be formed is omitted. See below.

According to the Standard, when initializing a reference to a non-volatile const class type (*cv1* T1) with an rvalue expression (*cv2* T2) where *cv1* T1 is reference compatible with *cv2* T2, the implementation shall proceed in one of the following ways (except when initializing the implicit object parameter of a copy constructor) 11.6.3 [dcl.init.ref] paragraph 5 bullet 2 sub-bullet 1:

- The reference is bound to the object represented by the rvalue (see 6.10 [basic.lval]) or to a sub-object within that object.
- A temporary of type "*cv1* T2" [sic] is created, and a constructor is called to copy the entire rvalue object into the temporary...

While the first case is quite obvious, the second one is a bit unclear as it says "a constructor is called to copy the entire rvalue object into the temporary" without specifying how the temporary is created -- by direct-initialization or by copy-initialization? As stated in DR 152, this can make a difference when the copy constructor is declared as explicit. How should the set of candidate functions be formed? The most appropriate guess is that it shall proceed as per 16.3.1.3 [over.match.ctor].

Another detail worth of note is that in the draft version of the Standard as of 2 December 1996 the second bullet read:

- A temporary of type "*cv1* T2" [sic] is created, and a copy constructor is called to copy the entire rvalue object into the temporary...

J. Stephen Adamczyk replied that the reason for changing "a copy constructor" to "a constructor" was to allow for member template converting constructors.

However, the new wording is somewhat in conflict with the footnote #93 that says that when initializing the implicit object parameter of a copy constructor an implementation must eventually choose the first alternative (binding without copying) to avoid infinite recursion. This seems to suggest that a copy constructor is always used for initializing the temporary of type "*cv1* T2".

Furthermore, now that the set of candidate functions is not limited to only the copy constructors of T2, there might be some unpleasant consequences. Consider a rather contrived sample below:

```
int   * pi = ::new(std::nothrow) int;
const std::auto_ptr<int>   & ri = std::auto_ptr<int>(pi);
```

In this example the initialization of the temporary of type '<TT>const std::auto_ptr<int>' (to which 'ri' is meant to be subsequently bound) doesn't fail, as it would had the approach with copy constructors been retained, instead, a yet another temporary gets created as the well-known sequence:

```
std::auto_ptr<int>::operator std::auto_ptr_ref<int>()
std::auto_ptr<int>(std::auto_ptr_ref<int>)
```

is called (assuming, of course, that the set of candidate functions is formed as per 16.3.1.3 [over.match.ctor]). The second temporary is transient and gets destroyed at the end of the initialization. I doubt that this is the way that the committee wanted this kind of reference binding to go.

Besides, even if the approach restricting the set of candidates to copy constructors is restored, it is still not clear how the initialization of the temporary (to which the reference is intended to be bound) is to be performed -- using direct-initialization or copy-initialization.

Another place in the Standard that would benefit from a similar clarification is the creation of an exception object, which is delineated in 18.1 [except.throw].

David Abrahams (February 2004): It appears, looking at core 291, that there may be a need to tighten up 11.6.3 [dcl.init.ref]/5.

Please see the attached example file, which demonstrates "move semantics" in C++98. Many compilers fail to compile test 10 because of the way 8.5.3/5 is interpreted. My problem with that interpretation is that test 20:

```
typedef X const XC;
sink2(XC(X()));
```

does compile. In other words, it *is* possible to construct the const temporary from the rvalue. IMO, that is the proper test.

8.5.3/5 doesn't demand that a "copy constructor" is used to copy the temporary, only that a constructor is used "to copy the temporary". I hope that when the language is tightened up to specify direct (or copy initialization), that it also unambiguously allows the enclosed test

to compile. Not only is it, I believe, within the scope of reasonable interpretation of the current standard, but it's an incredibly important piece of functionality for library writers and users alike.

```
#include <iostream>
#include <cassert>

template <class T, class X>
struct enable_if_same
{
};

template <class X>
struct enable_if_same<X, X>
{
    typedef char type;
};

struct X
{
    static int cnt;  // count the number of Xs

    X()
      : id(++cnt)
      , owner(true)
    {
        std::cout << "X() #" << id << std::endl;
    }

    // non-const lvalue - copy ctor
    X(X& rhs)
      : id(++cnt)
      , owner(true)
    {
        std::cout << "copy #" << id << " <- #" << rhs.id << std::endl;
    }

    // const lvalue - T will be deduced as X const
    template <class T>
    X(T& rhs, typename enable_if_same<X const,T>::type = 0)
      : id(++cnt)
      , owner(true)
    {
        std::cout << "copy #" << id << " <- #" << rhs.id << " (const)" << std::endl;
    }

    ~X()
    {
        std::cout << "destroy #" << id << (owner?"":" (EMPTY)") << std::endl;
    }

 private:    // Move stuff
    struct ref { ref(X*p) : p(p) {} X* p; };

 public:    // Move stuff
    operator ref() {
        return ref(this);
    }

    // non-const rvalue
    X(ref rhs)
      : id(++cnt)
      , owner(rhs.p->owner)
    {
        std::cout << "MOVE #" << id << " <== #" << rhs.p->id << std::endl;
        rhs.p->owner = false;
        assert(owner);
    }

 private:   // Data members
    int id;
    bool owner;
};

int X::cnt;


X source()
{
    return X();
}

X const csource()
{
    return X();
}

void sink(X)
{
    std::cout << "in rvalue sink" << std::endl;
}

void sink2(X&)
{
    std::cout << "in non-const lvalue sink2" << std::endl;
}

void sink2(X const&)
{
    std::cout << "in const lvalue sink2" << std::endl;
}

void sink3(X&)
{
```

```
    std::cout << "in non-const lvalue sink3" << std::endl;
}

template <class T>
void tsink(T)
{
    std::cout << "in templated rvalue sink" << std::endl;
}

int main()
{
    std::cout << " ------ test 1, direct init from rvalue ------- " << std::endl;
#ifdef __GNUC__ // GCC having trouble parsing the extra parens
    X z2((0, X() ));
#else
    X z2((X()));
#endif

    std::cout << " ------ test 2, copy init from rvalue ------- " << std::endl;
    X z4 = X();

    std::cout << " ------ test 3, copy init from lvalue ------- " << std::endl;
    X z5 = z4;

    std::cout << " ------ test 4, direct init from lvalue ------- " << std::endl;
    X z6(z4);

    std::cout << " ------ test 5, construct const ------- " << std::endl;
    X const z7;

    std::cout << " ------ test 6, copy init from lvalue ------- " << std::endl;
    X z8 = z7;

    std::cout << " ------ test 7, direct init from lvalue ------- " << std::endl;
    X z9(z7);

    std::cout << " ------ test 8, pass rvalue by-value ------- " << std::endl;
    sink(source());

    std::cout << " ------ test 9, pass const rvalue by-value ------- " << std::endl;
    sink(csource());

    std::cout << " ------ test 10, pass rvalue by overloaded reference ------- " << std::endl;
    // This one fails in Comeau's strict mode due to 8.5.3/5.  GCC 3.3.1 passes it.
    sink2(source());

    std::cout << " ------ test 11, pass const rvalue by overloaded reference ------- " << std::endl;
    sink2(csource());

#if 0    // These two correctly fail to compile, just as desired
    std::cout << " ------ test 12, pass rvalue by non-const reference ------- " << std::endl;
    sink3(source());

    std::cout << " ------ test 13, pass const rvalue by non-const reference ------- " << std::endl;
    sink3(csource());
#endif

    std::cout << " ------ test 14, pass lvalue by-value ------- " << std::endl;
    sink(z5);

    std::cout << " ------ test 15, pass const lvalue by-value ------- " << std::endl;
    sink(z7);

    std::cout << " ------ test 16, pass lvalue by-reference ------- " << std::endl;
    sink2(z4);

    std::cout << " ------ test 17, pass const lvalue by const reference ------- " << std::endl;
    sink2(z7);

    std::cout << " ------ test 18, pass const lvalue by-reference ------- " << std::endl;
#if 0   // correctly fails to compile, just as desired
    sink3(z7);
#endif

    std::cout << " ------ test 19, pass rvalue by value to template param ------- " << std::endl;
    tsink(source());

    std::cout << " ------ test 20, direct initialize a const A with an A ------- " << std::endl;
    typedef X const XC;
    sink2(XC(X()));
}
```

Proposed Resolution:

(As proposed by N1610 section 5, with editing.)

Change paragraph 5, second bullet, first sub-bullet, second sub-sub-bullet as follows:

> A temporary of type "*cv1* `T2`" [sic] is created~~, and a constructor is called to copy the entire rvalue object into the temporary~~ **via copy-initialization from the entire rvalue object**. The reference is bound to the temporary or to a sub-object within the temporary.

The text immediately following that is changed as follows:

> ~~The constructor that would be used to make the copy shall be callable whether or not the copy is actually done.~~ **The constructor and any conversion function that would be used in the initialization shall be callable whether or not the temporary is actually created.**

Note, however, that the way the core working group is leaning on issue 391 (i.e., requiring direct binding) would make this change unnecessary.

Proposed resolution (April, 2005):

This issue is resolved by the resolution of issue 391.

---

391. Require direct binding of short-lived references to rvalues

Section: 11.6.3 [dcl.init.ref] **Status:** CD1 **Submitter:** Raoul Gough **Date:** 14 Nov 2002

[Voted into WP at October 2005 meeting.]

After some email exchanges with Rani Sharoni, I've come up with the following proposal to allow reference binding to non-copyable rvalues in some cases. Rationale and some background appear afterwards.

---- proposal ----

Replace the section of 11.6.3 [dcl.init.ref] paragraph 5 that begins "If the initializer expression is an rvalue" with the following:

- If the initializer expression is an rvalue, with T2 a class type, and ``cv1 T1'' is reference-compatible with ``cv2 T2,'' the reference is bound as follows:
  - If the lifetime of the reference does not extend beyond the end of the full expression containing the initializer expression, the reference is bound to the object represented by the rvalue (see 6.10 [basic.lval]) or to a sub-object within that object.
  - otherwise, the reference is bound in one of the following ways (the choice is implementation-defined):
    - [... continues as before - the original wording applies unchanged to longer-lived references]

---- rationale ----

1. The intention of the current wording is to provide the implementation freedom to construct an rvalue of class type at an arbitrary location and copy it zero or more times before binding any reference to it.
2. The standard allows code to call a member function on an rvalue of class type (in 8.2.5 [expr.ref], I guess). This means that the implementation can be forced to bind the reference directly, with no freedom to create any temporary copies. e.g.

   ```
   class nc {
     nc (nc const &);  // private, nowhere defined
   public:
     nc ();
     nc const &by_ref () const { return *this; }
   };

   void f () {
     void g (nc const &);

     g (nc());          // Ill-formed
     g (nc().by_ref()); // Ok - binds directly to rvalue
   }
   ```

   Forcing a direct binding in this way is possible wherever the lifetime of the reference does not extend beyond the containing full expression, since the reference returned by the member function remains valid for this long.
3. As demonstrated above, existing implementations must already be capable of constructing an rvalue of class type in the "right" place the first time. Some compilers already silently allow the direct binding of references to non-copyable rvalues.
4. The change will not break any portable user code. It would break any platform-specific user code that relies on copies being performed by the particular implementation.

---- background ----

The proposal is based on a recent discussion in this group. I originally wanted to leave the implementation free to copy the rvalue if there was a callable copy constructor, and only **have** to bind directly if none was callable. Unfortunately, a traditional compiler can't always tell whether a function is callable or not, e.g. if the copy constructor is declared but not defined. Rani pointed this out in an example, and suggested that maybe trivial copy constructors should still be allowed (by extension, maybe wherever the compiler can determine callability). I've gone with this version because it's simpler, and I also figure the "as if" rule gives the compiler some freedom with POD types anyway.

Notes from April 2003 meeting:

We agreed generally with the proposal. We were unsure about the need for the restriction regarding long-lived references. We will check with the proposer about that.

Jason Merrill points out that the test case in issue 86 may be a case where we do not want to require direct binding.

Further information from Rani Sharoni (April 2003):

I wasn't aware about the latest suggestion of Raoul as it appears in core issue 391. In our discussions we tried to formulate a different proposal.

The rational, as we understood, behind the implementation freedom to make an extra copying (8.5.3/5/2/12) of the rvalue is to allow return values in registers which on some architectures are not addressable. The example that Raoul and I presented shows that this

implementation freedom is not always possible since we can "force" the rvalue to be addressable using additional member function (by_ref). The example only works for short lived rvalues and this is probably why Raoul narrow the suggestion.

I had different rational which was related to the implementation of conditional operator in VC. It seems that when conditional operator is involved VC does use an extra copying when the lifetime of the temporary is extended:

```
struct A { /* ctor with side effect */};

void f(A& x) {
  A const& r = cond ? A(1) : x; // VC actually make an extra copy of
                                // the rvalue A(1)
}
```

I don't know what the consideration behind the VC implementation was (I saw open bug on this issue) but it convinced me to narrow the suggestion.

IMHO such limitation seems to be too strict because it might limit the optimizer since returning class rvalues in registers might be useful (although I'm not aware about any implementation that actually does it). My suggestion was to forbid the extra copying if the ctor is not viable (e.g. A::A(A&)). In this case the implementation "freedom" doesn't exist (since the code might not compile) and only limits the programmer freedom (e.g. Move Constructors - http://www.cuj.com/experts/2102/alexandr.htm).

Core issue 291 is strongly related to the above issue and I personally prefer to see it resolved first. It seems that VC already supports the resolution I prefer.

Notes from October 2003 meeting:

We ended up feeling that this is just one of a number of cases of optimizations that are widely done by compilers and allowed but not required by the standard. We don't see any strong reason to require compilers to do this particular optimization.

Notes from the March 2004 meeting:

After discussing issue 450, we found ourselves reconsidering this, and we are now inclined to make a change to require the direct binding in all cases, with no restriction on long-lived references. Note that such a change would eliminate the need for a change for issue 291.

Proposed resolution (October, 2004):

Change 11.6.3 [dcl.init.ref] paragraph 5 bullet 2 sub-bullet 1 as follows:

> If the initializer expression is an rvalue, with `T2` a class type, and "*cv1* `T1`" is reference-compatible with "*cv2* `T2`", the reference is bound **to the object represented by the rvalue (see 6.10 [basic.lval]) or to a sub-object within that object.** ~~in one of the following ways (the choice is implementation-defined):~~
>
> - ~~The reference is bound to the object represented by the rvalue (see 6.10 [basic.lval]) or to a sub-object within that object.~~
> - ~~A temporary of type "*cv1* `T2`" [sic] is created, and a constructor is called to copy the entire rvalue object into the temporary. The reference is bound to the temporary or to a sub-object within the temporary.~~
>
> ~~The constructor that would be used to make the copy shall be callable whether or not the copy is actually done.~~ [*Example:*
>
> ```
> struct A { };
> struct B : public A { } b;
> extern B f();
> const A& rca = f ();  // Bound ~~Either bound~~ to the A sub-object of the B rvalue~~,~~
> ~~                    // or the entire B object is copied and the reference~~
> ~~                    // is bound to the A sub-object of the copy~~
> ```
>
> —*end example*]

[This resolution also resolves issue 291.]

---

450. Binding a reference to const to a cv-qualified array rvalue

Section: 11.6.3 [dcl.init.ref] **Status:** CD1 **Submitter:** Steve Adamczyk **Date:** 16 Jan 2004

[Voted into WP at October 2005 meeting.]

It's unclear whether the following is valid:

```
const int N = 10;
const int M = 20;
typedef int T;
void f(T const (&x)[N][M]){}

struct X {
        int i[10][20];
};

X g();

int main()
{
        f(g().i);
}
```

When you run this through 11.6.3 [dcl.init.ref], you sort of end up falling off the end of the standard's description of reference binding. The standard says in the final bullet of paragraph 5 that an array temporary should be created and copy-initialized from the rvalue array, which seems implausible.

I'm not sure what the right answer is. I think I'd be happy with allowing the binding in this case. We would have to introduce a special case like the one for class rvalues.

Notes from the March 2004 meeting:

g++ and EDG give an error. Microsoft (8.0 beta) and Sun accept the example. Our preference is to allow the direct binding (no copy). See the similar issue with class rvalues in issue 391.

Proposed resolution (October, 2004):

1. Insert a new bullet in 11.6.3 [dcl.init.ref] paragraph 5 bullet 2 before sub-bullet 2 (which begins, "Otherwise, a temporary of type '*cv1* `T1`' is created..."):

   > If the initializer expression is an rvalue, with `T2` an array type, and "*cv1* `T1`" is reference-compatible with "*cv2* `T2`", the reference is bound to the object represented by the rvalue (see 6.10 [basic.lval]).

2. Change 6.10 [basic.lval] paragraph 2 as follows:

   > An lvalue refers to an object or function. Some rvalue expressions — those of **(possibly cv-qualified)** class **or array type** ~~or cv-qualified class type~~ — also refer to objects.

---

175. Class name injection and base name access

Section: 12 [class] **Status:** CD1 **Submitter:** John Spicer **Date:** 21 February 1999

[Moved to DR at 10/01 meeting.]

With class name injection, when a base class name is used in a derived class, the name found is the injected name in the base class, not the name of the class in the scope containing the base class. Consequently, if the base class name is not accessible (e.g., because is is in a private base class), the base class name cannot be used unless a qualified name is used to name the class in the class or namespace of which it is a member.

Without class name injection the following example is valid. With class name injection, `A` is inaccessible in class `C`.

```
class A { };
class B: private A { };
class C: public B {
    A* p;    // error: A inaccessible
};
```

At the least, the standard should be more explicit that this is, in fact, ill-formed.

(See paper J16/99-0010 = WG21 N1187.)

Proposed resolution (04/01):

Add to the end of 14.1 [class.access.spec] paragraph 3:

> [*Note:* In a derived class, the lookup of a base class name will find the injected-class-name instead of the name of the base class in the scope in which it was declared. The injected-class-name might be less accessible than the name of the base class in the scope in which it was declared.] [*Example:*
>
> ```
> class A { };
> class B : private A { };
> class C : public B {
>     A* p;    // error: injected-class-name A is inaccessible
>     ::A* q;  // OK
> };
> ```
>
> —*end example*]

---

273. POD classes and `operator&()`

Section: 12 [class] **Status:** CD1 **Submitter:** Andrei Iltchenko **Date:** 10 Mar 2001

[Moved to DR at October 2002 meeting.]

I think that the definition of a POD class in the current version of the Standard is overly permissive in that it allows for POD classes for which a user-defined operator function `operator&` may be defined. Given that the idea behind POD classes was to achieve compatibility with C structs and unions, this makes 'Plain old' structs and unions behave not quite as one would expect them to.

In the C language, if `x` and `y` are variables of struct or union type `S` that has a member `m`, the following expression are allowed: `&x`, `x.m`, `x = y`. While the C++ standard guarantees that if `x` and `y` are objects of a POD class type `S`, the expressions `x.m`, `x = y` will have the same effect as they would in C, it is still possible for the expression `&x` to be interpreted differently, subject to the programmer supplying an appropriate version of a user-defined operator function `operator&` either as a member function or as a non-member function.

This may result in surprising effects. Consider:

```
// POD_bomb is a POD-struct. It has no non-static non-public data members,
// no virtual functions, no base classes, no constructors, no user-defined
// destructor, no user-defined copy assignment operator, no non-static data
// members of type pointer to member, reference, non-POD-struct, or
// non-POD-union.
struct  POD_bomb  {
   int   m_value1;
   int   m_value2;
   int  operator&()
   {   return  m_value1++;   }
   int  operator&() const
   {   return  m_value1 + m_value2;   }
};
```

6.9 [basic.types] paragraph 2 states:

> For any complete POD object type `T`, whether or not the object holds a valid value of type `T`, the underlying bytes (4.4 [intro.memory]) making up the object can be copied into an array of `char` or `unsigned char` [*footnote*: By using, for example, the library functions (20.5.1.2 [headers]) `memcpy` or `memmove`]. If the content of the array of `char` or `unsigned char` is copied back into the object, the object shall subsequently hold its original value. [*Example*:
>
> ```
> #define N sizeof(T)
> char buf[N];
> T obj;   // obj initialized to its original value
> memcpy(buf, &obj, N);
>             // between these two calls to memcpy,
>             // obj might be modified
> memcpy(&obj, buf, N);
>             // at this point, each subobject of obj of scalar type
>             // holds its original value
> ```
>
> —*end example*]

Now, supposing that the complete POD object type `T` in the example above is `POD_bomb`, and we cannot any more count on the assertions made in the comments to the example. Given a standard conforming implementation, the code will not even compile. And I see no legal way of copying the contents of an object of a complete object type `POD_bomb` into an array of `char` or `unsigned char` with `memcpy` or `memmove` without making use of the unary `&` operator. Except, of course, by means of an ugly construct like:

```
struct  POD_without_ampersand  {
   POD_bomb   a_bomb;
}  obj;
#define N sizeof(POD_bomb)
char buf[N];
memcpy(buf, &obj, N);
memcpy(&obj, buf, N);
```

The fact that the definition of a POD class allows for POD classes for which a user-defined `operator&` is defined, may also present major obstacles to implementers of the offsetof macro from <cstddef>

21.2 [support.types] paragraph 5 says:

> The macro `offsetof` accepts a restricted set of type arguments in this International Standard. `type` shall be a POD structure or a POD union (clause 12 [class]). The result of applying the `offsetof` macro to a field that is a static data member or a function is undefined."

Consider a well-formed C++ program below:

```
#include <cstddef>
#include <iostream>


struct  POD_bomb  {
   int   m_value1;
   int   m_value2;
   int  operator&()
   {   return  m_value1++;   }
   int  operator&() const
   {   return  m_value1 + m_value2;   }
};


// POD_struct is a yet another example of a POD-struct.
struct  POD_struct  {
   POD_bomb   m_nonstatic_bomb1;
   POD_bomb   m_nonstatic_bomb2;
};


int  main()
{

   std::cout << "offset of m_nonstatic_bomb2: " << offsetof(POD_struct,
       m_nonstatic_bomb2) << '\n';
   return  0;

}
```

See Jens Maurer's paper 01-0038=N1324 for an analysis of this issue.

Notes from 10/01 meeting:

A consensus was forming around the idea of disallowing `operator&` in POD classes when it was noticed that it is permitted to declare global-scope `operator&` functions, which cause the same problems. After more discussion, it was decided that such functions should not be prohibited in POD classes, and implementors should simply be required to "get the right answer" in constructs such as `offsetof` and `va_start` that are conventionally implemented using macros that use the "`&`" operator. It was noted that one can cast the original operand to `char &` to de-type it, after which one can use the built-in "`&`" safely.

Proposed resolution:

- Add a footnote in 21.2 [support.types] paragraph 5:

  [*Footnote:* Note that `offsetof` is required to work as specified even if unary `operator&` is overloaded for any of the types involved.]

- Add a footnote in 21.10 [support.runtime] paragraph 3:

  [*Footnote:* Note that `va_start` is required to work as specified even if unary `operator&` is overloaded for the type of `parmN`.]

---

284. *qualified-id*s in class declarations

Section: 12 [class] **Status:** CD1 **Submitter:** Mike Miller **Date:** 01 May 2001

[Moved to DR at October 2002 meeting.]

Although 11.3 [dcl.meaning] requires that a declaration of a *qualified-id* refer to a member of the specified namespace or class and that the member not have been introduced by a *using-declaration*, it applies only to names declared in a declarator. It is not clear whether there is existing wording enforcing the same restriction for *qualified-id*s in *class-specifier*s and *elaborated-type-specifier*s or whether additional wording is required. Once such wording is found/created, the proposed resolution of issue 275 must be modified accordingly.

Notes from 10/01 meeting:

The sentiment was that this should be required on class definitions, but not on elaborated type specifiers in general (which are references, not declarations). We should also make sure we consider explicit instantiations, explicit specializations, and friend declarations.

Proposed resolution (10/01):

Add after the end of 12.1 [class.name] paragraph 3:

> When a *nested-name-specifier* is specified in a *class-head* or in an *elaborated-type-specifier*, the resulting qualified name shall refer to a previously declared member of the class or namespace to which the *nested-name-specifier* refers, and the member shall not have been introduced by a using-declaration in the scope of the class or namespace nominated by the *nested-name-specifier*.

---

327. Use of "structure" without definition

Section: 12 [class] **Status:** CD1 **Submitter:** James Kanze **Date:** 9 Dec 2001

[Voted into WP at April, 2007 meeting.]

In 12 [class] paragraph 4, the first sentence says "A structure is a class definition defined with the *class-key* `struct`". As far as I know, there is no such thing as a structure in C++; it certainly isn't listed as one of the possible compound types in 6.9.2 [basic.compound]. And defining structures opens the question of whether a forward declaration is a structure or not. The parallel here with union (which follows immediately) suggests that structures and classes are really different things, since the same wording is used, and classes and unions do have some real differences, which manifest themselves outside of the definition. It also suggests that since one can't forward declare union with class and vice versa, the same should hold for struct and class -- I believe that the intent was that one could use struct and class interchangeably in forward declaration.

Suggested resolution:

I suggest something like the following:

> If a class is defined with the *class-key* `class`, its members and base classes are private by default. If a class is defined with the *class-key* `struct`, its members and base classes are public by default. If a class is defined with the *class-key* `union`, its members are public by default, and it holds only one data member at a time. Such classes are called unions, and obey a number of additional restrictions, see 12.3 [class.union].

Proposed resolution (April, 2006):

This issue is resolved by the resolution of issue 538.

379. Change "class declaration" to "class definition"

Section: 12 [class] **Status:** CD1 **Submitter:** Jens Maurer **Date:** 21 Oct 2002

[Voted into WP at March 2004 meeting.]

The ARM used the term "class declaration" to refer to what would usually be termed the definition of the class. The standard now often uses "class definition", but there are some surviving uses of "class declaration" with the old meaning. They should be found and changed.

Proposed resolution (April 2003):

Replace in 6.1 [basic.def] paragraph 2

> A declaration is a *definition* unless it declares a function without specifying the function's body (11.4 [dcl.fct.def]), it contains the `extern` specifier (10.1.1 [dcl.stc]) or a *linkage-specification* [Footnote: Appearing inside the braced-enclosed *declaration-seq* in a *linkage-specification* does not affect whether a declaration is a definition. --- end footnote] (10.5 [dcl.link]) and neither an *initializer* nor a *function-body*, it declares a static data member in a class ~~declaration~~ **definition** (12.2.3 [class.static]), it is a class name declaration (12.1 [class.name]), or it is a `typedef` declaration (10.1.3 [dcl.typedef]), a *using-declaration* (10.3.3 [namespace.udecl]), or a *using-directive* (10.3.4 [namespace.udir]).

Replace in 10.1.2 [dcl.fct.spec] paragraphs 5 and 6

> The `virtual` specifier shall only be used in declarations of nonstatic class member functions that appear within a *member-specification* of a class ~~declaration~~ **definition**; see 13.3 [class.virtual].
>
> The `explicit` specifier shall be used only in declarations of constructors within a class ~~declaration~~ **definition**; see 15.3.1 [class.conv.ctor].

Replace in 10.1.3 [dcl.typedef] paragraph 4

> A *typedef-name* that names a class is a *class-name* (12.1 [class.name]). If a *typedef-name* is used following the *class-key* in an *elaborated-type-specifier* (10.1.7.3 [dcl.type.elab]) or in the *class-head* of a class ~~declaration~~ **definition** (12 [class]), or is used as the *identifier* in the declarator for a constructor or destructor declaration (15.1 [class.ctor], 15.4 [class.dtor]), the program is ill-formed.

Replace in 10.3.1.2 [namespace.memdef] paragraph 3

> The name of the friend is not found by simple name lookup until a matching declaration is provided in that namespace scope (either before or after the class ~~declaration~~ **definition** granting friendship).

Replace in 11.3.2 [dcl.ref] paragraph 4

> There shall be no references to references, no arrays of references, and no pointers to references. The declaration of a reference shall contain an *initializer* (11.6.3 [dcl.init.ref]) except when the declaration contains an explicit `extern` specifier (10.1.1 [dcl.stc]), is a class member (12.2 [class.mem]) declaration within a class ~~declaration~~ **definition**, or is the declaration of a parameter or a return type (11.3.5 [dcl.fct]); see 6.1 [basic.def].

Replace in 11.6.3 [dcl.init.ref] paragraph 3

> The initializer can be omitted for a reference only in a parameter declaration (11.3.5 [dcl.fct]), in the declaration of a function return type, in the declaration of a class member within its class ~~declaration~~ **definition** (12.2 [class.mem]), and where the `extern` specifier is explicitly used.

Replace in 12.1 [class.name] paragraph 2

> A class ~~definition~~ **declaration** introduces the class name into the scope where it is ~~defined~~ **declared** and hides any class, object, function, or other declaration of that name in an enclosing scope (6.3 [basic.scope]). If a class name is declared in a scope where an object, function, or enumerator of the same name is also declared, then when both declarations are in scope, the class can be referred to only using an *elaborated-type-specifier* (6.4.4 [basic.lookup.elab]).

Replace in 12.2.3 [class.static] paragraph 4

> Static members obey the usual class member access rules (clause 14 [class.access]). When used in the declaration of a class member, the `static` specifier shall only be used in the member declarations that appear within the *member-specification* of the class ~~declaration~~ **definition**.

Replace in 12.2.5 [class.nest] paragraph 1

> A class can be ~~defined~~ **declared** within another class. A class ~~defined~~ **declared** within another is called a *nested* class. The name of a nested class is local to its enclosing class. The nested class is in the scope of its enclosing class. Except by using explicit pointers, references, and object names, declarations in a nested class can use only type names, static members, and enumerators from the enclosing class.

Replace in 12.4 [class.local] paragraph 1

A class can be ~~defined~~ **declared** within a function definition; such a class is called a *local* class. The name of a local class is local to its enclosing scope. The local class is in the scope of the enclosing scope, and has the same access to names outside the function as does the enclosing function. Declarations in a local class can use only type names, static variables, `extern` variables and functions, and enumerators from the enclosing scope.

Replace in 13 [class.derived] paragraph 1

... The *class-name* in a *base-specifier* shall not be an incompletely defined class (clause 12 [class]); this class is called a *direct base class* for the class being ~~declared~~ **defined**. During the lookup for a base class name, non-type names are ignored (6.3.10 [basic.scope.hiding]). If the name found is not a *class-name*, the program is ill-formed. A class `B` is a base class of a class `D` if it is a direct base class of `D` or a direct base class of one of `D`'s base classes. A class is an *indirect* base class of another if it is a base class but not a direct base class. A class is said to be (directly or indirectly) *derived* from its (direct or indirect) base classes. [*Note:* See clause 14 [class.access] for the meaning of *access-specifier*.] Unless ~~redefined~~ **redeclared** in the derived class, members of a base class are also considered to be members of the derived class. The base class members are said to be *inherited* by the derived class. Inherited members can be referred to in expressions in the same manner as other members of the derived class, unless their names are hidden or ambiguous (13.2 [class.member.lookup]). [*Note:* the scope resolution operator `::` (_N4567_.5.1.1 [expr.prim.general]) can be used to refer to a direct or indirect base member explicitly. This allows access to a name that has been ~~redefined~~ **redeclared** in the derived class. A derived class can itself serve as a base class subject to access control; see 14.2 [class.access.base]. A pointer to a derived class can be implicitly converted to a pointer to an accessible unambiguous base class (7.11 [conv.ptr]). An lvalue of a derived class type can be bound to a reference to an accessible unambiguous base class (11.6.3 [dcl.init.ref]).]

Replace in 13.1 [class.mi] paragraph 5

For another example,

```
class V { /* ... */ };
class A : virtual public V { /* ... */ };
class B : virtual public V { /* ... */ };
class C : public A, public B { /* ... */ };
```

for an object `c` of class type `C`, a single subobject of type `V` is shared by every base subobject of `c` that ~~is declared to have~~ **has** a `virtual` base class of type `V`.

Replace in the example in 13.2 [class.member.lookup] paragraph 6 (the whole paragraph was turned into a note by the resolution of core issue 39)

The names ~~defined~~ **declared** in `V` and the left hand instance of `W` are hidden by those in `B`, but the names ~~defined~~ **declared** in the right hand instance of `W` are not hidden at all.

Replace in 13.4 [class.abstract] paragraph 2

... A virtual function is specified *pure* by using a *pure-specifier* (12.2 [class.mem]) in the function declaration in the class ~~declaration~~ **definition**. ...

Replace in the footnote at the end of 14.2 [class.access.base] paragraph 1

[Footnote: As specified previously in clause 14 [class.access], private members of a base class remain inaccessible even to derived classes unless `friend` declarations within the base class ~~declaration~~ **definition** are used to grant access explicitly.]

Replace in _N3225_.11.3 [class.access.dcl] paragraph 1

The access of a member of a base class can be changed in the derived class by mentioning its *qualified-id* in the derived class ~~declaration~~ **definition**. Such mention is called an *access declaration*. ...

Replace in the example in 16.4 [over.over] paragraph 5

The initialization of `pfe` is ill-formed because no `f()` with type `int(...)` has been ~~defined~~ **declared**, and not because of any ambiguity.

Replace in C.1.6 [diff.dcl] paragraph 1

Rationale: Storage class specifiers don't have any meaning when associated with a type. In C++, class members can be ~~defined~~ **declared** with the `static` storage class specifier. Allowing storage class specifiers on type declarations could render the code confusing for users.

Replace in C.1.8 [diff.class] paragraph 3

In C++, a typedef name may not be ~~redefined~~ **redeclared** in a class ~~declaration~~ **definition** after being used in ~~the declaration~~ **that definition**

Drafting notes:

The resolution of core issue 45 (DR) deletes 14.7 [class.access.nest] paragraph 2.

The following occurrences of "class declaration" are not changed:

- 8.3.4 [expr.new] paragraph 4
- 10.1.3 [dcl.typedef] paragraph 1
- 12.1 [class.name] paragraphs 3 and 4
- 14.3 [class.friend] paragraph 9

- C.1.8 [diff.class] paragraph 1

---

383. Is a class with a declared but not defined destructor a POD?

Section: 12 [class] **Status:** CD1 **Submitter:** Gennaro Prota **Date:** 18 Sep 2002

[Voted into WP at March 2004 meeting.]

The standard (12 [class] par. 4) says that "A POD-struct is an aggregate class that has no non-static data members of type non-POD-struct, non-POD-union (or array of such types) or reference, and has no user-defined copy assignment operator and no user-defined destructor."

Note that it says 'user-defined', not 'user-declared'. Is it the intent that if e.g. a copy assignment operator is declared but not defined, this does not (per se) prevent the class to be a POD-struct?

Proposed resolution (April 2003):

Replace in 12 [class] paragraph 4

> A *POD-struct* is an aggregate class that has no non-static data members of type non-POD-struct, non-POD-union (or array of such types) or reference, and has no user-~~defined~~ **declared** copy assignment operator and no user-~~defined~~ **declared** destructor. Similarly, a *POD-union* is an aggregate union that has no non-static data members of type non-POD-struct, non-POD-union (or array of such types) or reference, and has no user-~~defined~~ **declared** copy assignment operator and no user-~~defined~~ **declared** destructor.

Drafting note: The changes are shown relative to TC1, incorporating the changes from the resolution of core issue 148.

---

413. Definition of "empty class"

Section: 12 [class] **Status:** CD1 **Submitter:** Pete Becker **Date:** 30 Apr 2003

[Voted into WP at April, 2007 meeting.]

The proposal says that value is true if "T is an empty class (10)". Clause 10 doesn't define an empty class, although it has a note that says a base class may "be of zero size (clause 9)" 9/3 says "Complete objects and member subobjects of class type shall have nonzero size." This has a footnote, which says "Base class subobjects are not so constrained."

The standard uses the term "empty class" in two places (11.6.1 [dcl.init.aggr]), but neither of those places defines it. It's also listed in the index, which refers to the page that opens clause 9, i.e. the nonzero size stuff cited above.

So, what's the definition of "empty class" that determines whether the predicate is_empty is true?

The one place where it's used is 11.6.1 [dcl.init.aggr] paragraph 8, which says (roughly paraphrased) that an aggregate initializer for an empty class must be "{}", and when such an initializer is used for an aggregate that is not an empty class the members are default-initialized. In this context it's pretty clear what's meant. In the type traits proposal it's not as clear, and it was probably intended to have a different meaning. The boost implementation, after it eliminates non-class types, determines whether the trait is true by comparing the size of a class derived from T to the size of an otherwise-identical class that is not derived from T.

Howard Hinnant: `is_empty` was created to find out whether a type could be derived from and have the empty base class optimization successfully applied. It was created in part to support `compressed_pair` which attempts to optimize away the space for one of its members in an attempt to reduce spatial overhead. An example use is:

```
template <class T, class Compare = std::less<T> >
class SortedVec
{
public:
...
private:
  T* data_;
  compressed_pair<Compare, size_type> comp_;

  Compare&       comp()       {return comp_.first();}
  const Compare& comp() const {return comp_.first();}
  size_type&     sz()         {return comp_.second();}
  size_type      sz() const   {return comp_.second();}
};
```

Here the compare function is optimized away via the empty base optimization if Compare turns out to be an "empty" class. If Compare turns out to be a non-empty class, or a function pointer, the space is not optimized away. `is_empty` is key to making this work.

This work built on Nathan's article: http://www.cantrip.org/emptyopt.html.

Clark Nelson: I've been looking at issue 413, and I've reached the conclusion that there are two different kinds of empty class. A class containing only one or more anonymous bit-field members is empty for purposes of aggregate initialization, but not (necessarily) empty for purposes of empty base-class optimization.

Of course we need to add a definition of emptiness for purposes of aggregate initialization. Beyond that, there are a couple of questions:

1. Should the definition of emptiness used by the is_empty predicate be defined in a language clause or a library clause?
2. Do we need to open a new core issue pointing out the fact that the section on aggregate initialization does not currently say that unnamed bit-fields are skipped?

Notes from the October, 2005 meeting:

There are only two places in the Standard where the phrase "empty class" appears, both in 11.6.1 [dcl.init.aggr] paragraph 8. Because it is not clear whether the definition of "empty for initialization purposes" is suitable for use in defining the `is_empty` predicate, it would be better just to avoid using the phrase in the language clauses. The requirements of 11.6.1 [dcl.init.aggr] paragraph 8 appear to be redundant; paragraph 6 says that an *initializer-list* must have no more initializers than the number of elements to initialize, so an empty class already requires an empty *initializer-list*, and using an empty *initializer-list* with a non-empty class is covered adequately by paragraph 7's description of the handling of an *initializer-list* with fewer *initializer*s than the number of members to initialize.

Proposed resolution (October, 2005):

1. Change 11.6.1 [dcl.init.aggr] paragraph 5 by inserting the indicated text:

   > Static data members **and anonymous bit fields** are not considered members of the class for purposes of aggregate initialization. [*Example:*
   >
   > ```
   > struct A {
   >     int i;
   >     static int s;
   >     int j;
   >      int :17;
   >      int k;
   > } a = { 1, 2, 3 };
   > ```
   >
   > Here, the second initializer 2 initializes `a.j` and not the static data member `A::s`**, and the third initializer 3 initializes** `a.k` **and not the padding before it.** —*end example*]

2. Delete 11.6.1 [dcl.init.aggr] paragraph 8:

   > An *initializer* for an aggregate member that is an empty class shall have the form of an empty *initializer-list* `{}`. [*Example:*
   >
   > ```
   > struct S { };
   > struct A {
   >     S s;
   >     int i;
   > } a = { { }, 3 };
   > ```
   >
   > —*end example*] An empty initializer-list can be used to initialize any aggregate. If the aggregate is not an empty class, then each member of the aggregate shall be initialized with a value of the form `T()` (8.2.3 [expr.type.conv]), where `T` represents the type of the uninitialized member.

This resolution also resolves issue 491.

Additional note (October, 2005):

Deleting 11.6.1 [dcl.init.aggr] paragraph 8 altogether may not be a good idea. It would appear that, in its absence, the initializer elision rules of paragraph 11 would allow the initializer for `a` in the preceding example to be written `{ 3 }` (because the empty-class member `s` would consume no *initializer*s from the list).

Proposed resolution (October, 2006):

(Drafting note: this resolution also cleans up incorrect references to syntactic non-terminals in the nearby paragraphs.)

1. Change 11.6.1 [dcl.init.aggr] paragraph 4 as indicated:

   > An array of unknown size initialized with a brace-enclosed *initializer-list* containing `n` ~~*initializers*~~ ***initializer-clauses***, where `n` shall be greater than zero... An empty initializer list `{}` shall not be used as the ~~initializer~~ ***initializer-clause*** for an array of unknown bound.

2. Change 11.6.1 [dcl.init.aggr] paragraph 5 by inserting the indicated text:

   > Static data members **and anonymous bit fields** are not considered members of the class for purposes of aggregate initialization. [*Example:*
   >
   > ```
   > struct A {
   >     int i;
   >     static int s;
   >     int j;
   >      int :17;
   >      int k;
   > } a = { 1, 2, 3 };
   > ```
   >
   > Here, the second initializer `2` initializes `a.j` and not the static data member `A::s`**, and the third initializer** `3` **initializes** `a.k` **and not the anonymous bit field before it.** —*end example*]

3. Change 11.6.1 [dcl.init.aggr] paragraph 6 as indicated:

   > An *initializer-list* is ill-formed if the number of ~~*initializers*~~ ***initializer-clauses*** exceeds the number of members...

4. Change 11.6.1 [dcl.init.aggr] paragraph 7 as indicated:

If there are fewer ~~*initializers*~~ ***initializer-clauses*** in the list than there are members...

5. Replace 11.6.1 [dcl.init.aggr] paragraph 8:

   An *initializer* for an aggregate member that is an empty class shall have the form of an empty *initializer-list* `{}`. [*Example:*

   ```
   struct S { };
   struct A {
       S s;
       int i;
   } a = { { } , 3 };
   ```

   —*end example*] An empty initializer-list can be used to initialize any aggregate. If the aggregate is not an empty class, then each member of the aggregate shall be initialized with a value of the form `T()` (8.2.3 [expr.type.conv]), where `T` represents the type of the uninitialized member.

   with:

   If an aggregate class `C` contains a subaggregate member `m` that has no members for purposes of aggregate initialization, the *initializer-clause* for `m` shall not be omitted from an *initializer-list* for an object of type `C` unless the *initializer-clause*s for all members of `C` following `m` are also omitted. [*Example:*

   ```
   struct S { } s;
   struct A {
       S s1;
       int i1;
       S s2;
       int i2;
       S s3;
       int i3;
   } a = {
       { },    // Required initialization
       0,
       s,      // Required initialization
       0
   };          // Initialization not required for A::s3 because A::i3 is also not initialized
   ```

   —*end example*]

6. Change 11.6.1 [dcl.init.aggr] paragraph 10 as indicated:

   When initializing a multi-dimensional array, the ~~*initializers*~~ ***initializer-clauses*** initialize the elements...

7. Change 11.6.1 [dcl.init.aggr] paragraph 11 as indicated:

   Braces can be elided in an *initializer-list* as follows. If the *initializer-list* begins with a left brace, then the succeeding comma-separated list of ~~*initializers*~~ ***initializer-clauses*** initializes the members of a subaggregate; it is erroneous for there to be more ~~initializers~~ ***initializer-clauses*** than members. If, however, the *initializer-list* for a subaggregate does not begin with a left brace, then only enough ~~*initializers*~~ ***initializer-clauses*** from the list are taken to initialize the members of the subaggregate; any remaining ~~*initializers*~~ ***initializer-clauses*** are left to initialize the next member of the aggregate of which the current subaggregate is a member. [*Example:*...

8. Change 11.6.1 [dcl.init.aggr] paragraph 12 as indicated:

   All implicit type conversions (clause 7 [conv]) are considered when initializing the aggregate member with an ~~initializer from an *initializer-list*~~ ***assignment-expression***. If the ~~*initializer*~~ ***assignment-expression*** can initialize a member, the member is initialized. Otherwise, if the member is itself a ~~non-empty~~ subaggregate, brace elision is assumed and the ~~*initializer*~~ ***assignment-expression*** is considered for the initialization of the first member of the subaggregate. **[*Note:* As specified above, brace elision cannot apply to subaggregates with no members for purposes of aggregate initialization; an *initializer-clause* for the entire subobject is required. —*end note*]** [*Example:*... Braces are elided around the ~~*initializer*~~ ***initializer-clause*** for `b.a1.i`...

9. Change 11.6.1 [dcl.init.aggr] paragraph 15 as indicated:

   When a union is initialized with a brace-enclosed initializer, the braces shall only contain an ~~initializer~~ ***initializer-clause*** for the first member of the union...

10. Change 11.6.1 [dcl.init.aggr] paragraph 16 as indicated:

    [*Note:* ~~as~~ **As** described above, the braces around the ~~initializer~~ ***initializer-clause*** for a union member can be omitted if the union is a member of another aggregate. —*end note*]

(Note: this resolution also resolves issue 491.)

---

538. Definition and usage of *structure*, *POD-struct*, *POD-union*, and *POD class*

Section: 12 [class] **Status:** CD1 **Submitter:** Alisdair Meredith **Date:** 10 August 2005

[Voted into WP at April, 2007 meeting.]

There are several problems with the terms defined in 12 [class] paragraph 4:

A *structure* is a class defined with the *class-key* `struct`; its members and base classes (clause 13 [class.derived]) are public by default (clause 14 [class.access]). A *union* is a class defined with the *class-key* `union`; its members are public by default and it holds only one data member at a time (12.3 [class.union]). [*Note:* aggregates of class type are described in 11.6.1 [dcl.init.aggr]. —*end note*] A *POD-struct* is an aggregate class that has no non-static data members of type non-POD-struct, non-POD-union (or array of such types) or reference, and has no user-declared copy assignment operator and no user-declared destructor. Similarly, a *POD-union* is an aggregate union that has no non-static data members of type non-POD-struct, non-POD-union (or array of such types) or reference, and has no user-declared copy assignment operator and no user-declared destructor. A *POD class* is a class that is either a POD-struct or a POD-union.

1. Although the term *structure* is defined here, it is used only infrequently throughout the Standard, often apparently inadvertently and consequently incorrectly:
   - 8.2.5 [expr.ref] paragraph 4: the use is in a note and is arguably correct and helpful.
   - 12.2 [class.mem] paragraph 11: the term is used (three times) in an example. There appears to be no reason to use it instead of "class," but its use is not problematic.
   - 20.3 [definitions] "**iostream class templates**:" the `traits` argument to the iostream class templates is (presumably unintentionally) constrained to be a structure, i.e., to use the `struct` keyword and not the `class` keyword in its definition.
   - B [implimits] paragraph 2: the minimum number of declarator operators is given for structures and unions but not for classes defined using the `class` keyword.
   - B [implimits] paragraph 2: class, structure, and union are used as disjoint terms in describing nesting levels. (The nonexistent nonterminal *struct-declaration-list* is used, as well.)

   There does not appear to be a reason for defining the term *structure*. The one reference where it is arguably useful, in the note in 8.2.5 [expr.ref], could be rewritten as something like, "'class objects' may be defined using the `class`, `struct`, or `union` *class-keys*; see clause 12 [class]."
2. Based on its usage later in the paragraph and elsewhere, "POD-struct" appears to be intended to exclude unions. However, the definition of "aggregate class" in 11.6.1 [dcl.init.aggr] paragraph 1 includes unions. Furthermore, the name itself is confusing, leading to the question of whether it was intended that only classes defined using `struct` could be POD-structs or if `class`-classes are included. The definition should probably be rewritten as, "A *POD-struct* is an aggregate class **defined with the *class-key*** `struct` **or the *class-key*** `class` that has no...
3. In most references outside clause 12 [class], POD-struct and POD-union are mentioned together and treated identically. These references should be changed to refer to the unified term, "POD class."
4. Noted in passing: 21.2 [support.types] paragraph 4 refers to the undefined terms "POD structure" and (unhyphenated) "POD union;" the pair should be replaced by a single reference to "POD class."

Proposed resolution (April, 2006):

1. Change 12 [class] paragraph 4 as indicated:

   ~~A *structure* is a class defined with the *class-key* `struct`; its members and base classes (clause 13 [class.derived]) are public by default (clause 14 [class.access]).~~ A *union* is a class defined with the *class-key* `union`; ~~its members are public by default and~~ it holds only one data member at a time (12.3 [class.union]). [*Note:* aggregates of class type are described in 11.6.1 [dcl.init.aggr]. —*end note*] ~~A *POD-struct* is an aggregate class that has no non-static data members of type non-POD-struct, non-POD-union (or array of such types) or reference, and has no user-declared copy assignment operator and no user-declared destructor. Similarly, a *POD-union* is an aggregate union that has no non-static data members of type non-POD-struct, non-POD-union (or array of such types) or reference, and has no user-declared copy assignment operator and no user-declared destructor. A *POD class* is a class that is either a POD-struct or a POD-union.~~ **A *POD class* is an aggregate class that has no non-static data members of non-POD type (or array of such a type) or reference, and has no user-declared copy assignment operator and no user-declared destructor. A *POD-struct* is a POD class defined with the *class-key*** `struct` **or the *class-key*** `class`**. A *POD-union* is a POD class defined with the *class-key*** `union`**.**
2. Change 14.2 [class.access.base] paragraph 2 as indicated:

   In the absence of an *access-specifier* for a base class, `public` is assumed when the derived class is ~~declared~~ **defined with the *class-key*** `struct` and `private` is assumed when the class is ~~declared~~ **defined with the *class-key*** `class`. [*Example:*...
3. Delete the note in 8.2.5 [expr.ref] paragraph 4:

   ~~[*Note:* "class objects" can be structures (12.2 [class.mem]) and unions (12.3 [class.union]). Classes are discussed in clause 12 [class]. —*end note*]~~
4. Change the commentary in the example in 12.2 [class.mem] paragraph 11 as indicated:

   ...an integer, and two pointers to ~~similar structures~~ **objects of the same type**. Once this definition...

   ...the `count` member of the ~~structure~~ **object** to which `sp` points; `s.left` refers to the `left` subtree pointer of the ~~structure~~ **object** `s`; and...
5. Change 20.3 [definitions] "**iostream class templates**" as indicated:

   ...the argument `traits` is a ~~structure~~ **class** which defines additional characteristics...
6. Change 21.6 [support.dynamic] paragraph 4 as indicated:

If *type* is not a ~~POD-structure or a POD-union~~ **POD class** (clause 9), the results are undefined.

7. Change the third bullet of B [implimits] paragraph 2 as indicated:

   - Pointer, array, and function declarators (in any combination) modifying ~~an~~ **a class,** arithmetic, ~~structure, union,~~ or incomplete type in a declaration [256].

8. Change the nineteenth bullet of B [implimits] paragraph 2 as indicated:

   - Data members in a single class~~, structure, or union~~ [16 384].

9. Change the twenty-first bullet of B [implimits] paragraph 2 as indicated:

   - Levels of nested class~~, structure, or union~~ definitions in a single ~~*struct-declaration-list*~~ ***member-specification*** [256].

10. Change C.6 [diff.library] paragraph 6 as indicated:

    The C++ Standard library provides 2 standard ~~structures~~ `struct`**s** from the C library, as shown in Table 126.

11. Change the last sentence of 6.9 [basic.types] paragraph 10 as indicated:

    Scalar types, ~~POD-struct types, POD-union types~~ **POD classes** (clause 12 [class]), arrays of such types and *cv-qualified* versions of these types (6.9.3 [basic.type.qualifier]) are collectively called *POD types*.

Drafting note: Do not change 6.9 [basic.types] paragraph 11, because it's a note and the definition of "layout-compatible" is separate for POD-struct and POD-union in 12.2 [class.mem].

(This resolution also resolves issue 327.)

---

568. Definition of POD is too strict

Section: 12 [class] **Status:** CD1 **Submitter:** Matt Austern **Date:** 20 March 2006

[Voted into the WP at the July, 2007 meeting as part of paper J16/07-0202 = WG21 N2342.]

A POD struct (12 [class] paragraph 4) is "an aggregate class that has no non-static data members of type non-POD-struct, non-POD-union (or array of such types), or reference, and that has no user-defined copy assignment operator and no user-defined destructor." Meanwhile, an aggregate class (11.6.1 [dcl.init.aggr] paragraph 1) must have "no user-declared constructors, no private or protecte non-static data members, no base classes, and no virtual functions."

This is too strict. The whole reason we define the notion of POD is for the layout compatibility guarantees in 12.2 [class.mem] paragraphs 14-17 and the byte-for-byte copying guarantees of 6.9 [basic.types] paragraph 2. None of those guarantees should be affected by the presence of ordinary constructors, any more than they're affected by the presence of any other member function. It's silly for the standard to make layout and memcpy guarantees for this class:

```
struct A {
   int n;
};
```

but not for this one:

```
struct B {
  int n;
  B(n_) : n(n_) { }
};
```

With either `A` or `B`, it ought to be possible to save an array of those objects to disk with a single call to Unix's write(2) system call or the equivalent. At present the standard says that it's legal for `A` but not `B`, and there isn't any good reason for that distinction.

Suggested resolution:

The following doesn't fix all problems (in particular it still doesn't let us treat `pair<int, int>` as a POD), but at least it goes a long way toward fixing the problem: in 11.6.1 [dcl.init.aggr] paragraph 1, change "no user-declared constructors" to "no nontrivial default constructor and no user-declared copy constructor."

(Yes, I'm aware that this proposed change would also allow brace initialization for some types that don't currently allow it. I consider this to be a feature, not a bug.)

Mike Miller: I agree that something needs to be done about "POD," but I'm not sure that this is it. My own take is that "POD" is used for too many different things — things that are related but not identical — and the concept should be split. The current definition is useful, as is, for issues regarding initialization and lifetime. For example, I wouldn't want to relax the prohibition of jumping over a constructor call in 9.7 [stmt.dcl] (which is currently phrased in terms of POD types). On the other hand, I agree that the presence of a user-declared constructor says nothing about layout and bitwise copying. This needs (IMHO) a non-trivial amount of further study to determine how many categories we need (instead of just POD versus non-POD), which guarantees and prohibitions go with which category, the interaction of "memcpy initialization" (for want of a better term) with object lifetime, etc.

(See paper J16/06-0172 = WG21 N2102.)

Proposed resolution (April, 2007):

Adoption of the POD proposal (currently J16/07-0090 = WG21 N2230) will resolve this issue.

---

417. Using derived-class qualified name in out-of-class nested class definition

Section: 12.1 [class.name] **Status:** CD1 **Submitter:** Jon Caves **Date:** 19 May 2003

[Voted into WP at October 2004 meeting.]

We had a user complain that our compiler was allowing the following code:

```
struct B {
  struct S;
};

struct D : B { };

struct D::S {
};
```

We took one look at the code and made the reasonable (I would claim) assumption that this was indeed a bug in our compiler. Especially as we had just fixed a very similar issue with the definition of static data members.

Imagine our surprise when code like this showed up in Boost and that every other compiler we tested accepts this code. So is this indeed legal (it seems like it must be) and if so is there any justification for this beyond 6.4.3.1 [class.qual]?

John Spicer: The equivalent case for a member function is covered by the declarator rules in 11.3 [dcl.meaning] paragraph 1. The committee has previously run into cases where a restriction should apply to both classes and non-classes, but fails to do so because there is no equivalent of 11.3 [dcl.meaning] paragraph 1 for classes.

Given that, by the letter of the standard, I would say that this case is allowed.

Notes from October 2003 meeting:

We feel this case should get an error.

Proposed Resolution (October 2003):

Note that the change here interacts with issue 432.

Add the following as a new paragraph immediately following 6.3.2 [basic.scope.pdecl] paragraph 2:

> The point of declaration for a class first declared by a *class-specifier* is immediately after the *identifier* or *template-id* (if any) in its *class-head* (Clause 12 [class]). The point of declaration for an enumeration is immediately after the *identifier* (if any) in its *enum-specifier* (10.2 [dcl.enum]).

Change point 1 of 6.3.7 [basic.scope.class] paragraph 1 to read:

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

[Note that the preceding change duplicates one of the changes in the proposed resolution of issue 432.]

Change 17.8.2 [temp.explicit] paragraph 2 to read:

> If the explicit instantiation is for a member function, a member class or a static data member of a class template specialization, the name of the class template specialization in the *qualified-id* for the member ~~declarator~~ **name** shall be a *template-id*.

Add the following as paragraph 5 of Clause 12 [class]:

> If a *class-head* contains a *nested-name-specifier*, the *class-specifier* shall refer to a class that was previously declared directly in the class or namespace to which the *nested-name-specifier* refers (i.e., neither inherited nor introduced by a *using-declaration*), and the *class-specifier* shall appear in a namespace enclosing the previous declaration.

Delete 12.1 [class.name] paragraph 4 (this was added by issue 284):

> When a *nested-name-specifier* is specified in a *class-head* or in an *elaborated-type-specifier*, the resulting qualified name shall refer to a previously declared member of the class or namespace to which the *nested-name-specifier* refers, and the member shall not have been introduced by a using-declaration in the scope of the class or namespace nominated by the *nested-name-specifier*.

---

328. Missing requirement that class member types be complete

Section: 12.2 [class.mem] **Status:** CD1 **Submitter:** Michiel Salters **Date:** 10 Dec 2001

[Voted into WP at March 2004 meeting.]

Is it legal to use an incomplete type (6.9 [basic.types] paragraph 6) as a class member, if no object of such class is ever created ?

And as a class template member, even if the template is instantiated, but no object of the instantiated class is created?

The consensus seems to be NO, but no wording was found in the standard which explicitly disallows it.

The problem seems to be that most of the restrictions on incomplete types are on their use in objects, but class members are not objects.

A possible resolution, if this is considered a defect, is to add to 6.2 [basic.def.odr] paragraph 4, (situations when T must be complete), the use of T as a member of a class or instantiated class template.

The thread on comp.std.c++ which brought up the issue was "Compiler differences: which is correct?", started 2001 11 30. <3c07c8fb$0$8507$ed9e5944@reading.news.pipex.net>

Proposed Resolution (April 2002, revised April 2003):

Change the first bullet of the note in 6.2 [basic.def.odr] paragraph 4 and add two new bullets following it, as follows:

- an object of type `T` is defined (6.1 [basic.def]~~, 8.3.4 [expr.new]~~), or
- a non-static class data member of type `T` is declared (12.2 [class.mem]), or
- `T` is used as the object type or array element type in a *new-expression* (8.3.4 [expr.new]), or

Replace 12.2 [class.mem] paragraph 8 by:

> Non-`static` (12.2.3 [class.static]) data members shall not have incomplete types. In particular, a class `C` shall not contain a non-`static` member of class `C`, but it can contain a pointer or reference to an object of class `C`.

See also 6.9 [basic.types] paragraph 6, which is relevant but not changed by the Proposed Resolution.

---

437. Is type of class allowed in member function exception specification?

Section: 12.2 [class.mem] **Status:** CD1 **Submitter:** Cary Coutant **Date:** 10 Oct 2003

[Voted into WP at April 2005 meeting.]

I've encountered a C++ program in which a member function wants to declare that it may throw an object of its own class, e.g.:

```
class Foo {
private:
   int val;
public:
   Foo( int &initval ) { val = initval; };
   void throwit() throw(Foo) { throw (*this); };
};
```

The compiler is complaining that Foo is an incomplete type, and can't be used in the exception specification.

My reading of the standard [basic.types] is inconclusive. Although it does state that the class declaration is considered complete when the closing brace is read, I believe it also intends that the member function declarations should not be semantically validated until the class has been completely declared.

If this isn't allowed, I don't know how else a member function could be declared to throw an object of its own class.

John Spicer: The type is considered complete within function bodies, but not in their declaration (see 12.2 [class.mem] paragraph 2).

Proposed Resolution:

Change 12.2 [class.mem] paragraph 2 as follows:

> Within the class *member-specification*, the class is regarded as complete within function bodies, default arguments***, exception-specification*s,** and constructor *ctor-initializer*s (including such things in nested classes).

Rationale: Taken with 11.3.5 [dcl.fct] paragraph 6, the *exception-specification* is the *only* part of a function declaration/definition in which the class name cannot be used because of its putative incompleteness. There is no justification for singling out exception specifications this way; both in the function body and in a `catch` clause, the class type will be complete, so there is no harm in allowing the class name to be used in the *exception-specification*.

---

613. Unevaluated uses of non-static class members

Section: 12.2 [class.mem] **Status:** CD1 **Submitter:** Herb Sutter **Date:** 28 October 2006

[Voted into WP at April, 2007 meeting.]

According to 12.2 [class.mem] paragraph 9, the name of a non-static data member can only be used with an object reference (explicit or implied by the `this` pointer of a non-static member function) or to form a pointer to member. This restriction applies even in the operand of `sizeof`, although the operand is not evaluated and thus no object is needed to perform the operation. Consequently, determining the size of a non-static class member often requires a circumlocution like

```
sizeof ((C*) 0)->m
```

instead of the simpler and more obvious (but incorrect)

```
sizeof (C::m)
```

The CWG considered this question as part of issue 198 and decided at that time to retain the restriction on consistency grounds: the rule was viewed as applying uniformly to expressions, and making an exception for `sizeof` would require introducing a special-purpose "wart."

The issue has recently resurfaced, in part due to the fact that the restriction would also apply to the `decltype` operator. Like the unary `&` operator to form a pointer to member, `sizeof` and `decltype` need neither an lvalue nor an rvalue, requiring solely the declarative information of the named operand. One possible approach would be to define the concept of "unevaluated operand" or the like, exempt unevaluated operands from the requirement for an object reference in 12.2 [class.mem] paragraph 9, and then define the operands of these operators as "unevaluated."

Proposed resolution (April, 2007):

The wording is given in paper J16/07-0113 = WG21 N2253.

---

620. Declaration order in layout-compatible POD structs

Section: 12.2 [class.mem] **Status:** CD1 **Submitter:** Martin Sebor **Date:** 1 March 2007

[Voted into the WP at the July, 2007 meeting as part of paper J16/07-0202 = WG21 N2342.]

It should be made clear in 12.2 [class.mem] paragraph 15,

> Two POD-struct (clause 12 [class]) types are layout-compatible if they have the same number of non-static data members, and corresponding non-static data members (in order) have layout-compatible types (6.9 [basic.types]).

that "corresponding... (in order)" refers to declaration order and not the order in which the members are laid out in memory.

However, this raises the point that, in cases where an *access-specifier* is involved, the declaration and layout order can be different (see paragraph 12). Thus, for two POD-struct classes `A` and `B`,

```
struct A {
    char c;
    int i;
}
struct B {
    char c;
  public:
    int i;
};
```

a compiler could move `B::i` before `B::c`, but `A::c` must precede `A::i`. It does not seem reasonable that these two POD-structs would be considered layout-compatible, even though they satisfy the requirement that corresponding members in declaration order are layout-compatible.

One possibility would be to require that neither POD-struct have an *access-specifier* in order to be considered layout-compatible. (It's not sufficient to require that they have the same *access-specifier*s, because the compiler is not required to lay out the storage the same way for different classes.)

11.6.1 [dcl.init.aggr] paragraph 2 should also be clarified to make explicit that "increasing... member order" refers to declaration order.

Proposed resolution (April, 2007):

This issue will be resolved by the adoption of the POD proposal (currently J16/07-0090 = WG21 N2230). That paper does not propose a change to the wording of 11.6.1 [dcl.init.aggr] paragraph 2, but the CWG feels that the intent of that paragraph (that the initializers are used in declaration order) is clear enough not to require revision.

---

452. Wording nit on description of `this`

Section: 12.2.2.1 [class.this] **Status:** CD1 **Submitter:** Gennaro Prota **Date:** 8 Jan 2004

[Voted into WP at July, 2007 meeting.]

12.2.2.1 [class.this] paragraph 1, which specifies the meaning of the keyword 'this', seems to limit its usage to the *body* of non-static member functions. However 'this' is also usable in ctor-initializers which, according to the grammar in 11.4 [dcl.fct.def] par. 1, are not part of the body.

Proposed resolution: Changing the first part of 12.2.2.1 [class.this] par. 1 to:

> In the body of a nonstatic (9.3) member function **or in a ctor-initializer (12.6.2)**, the keyword `this` is a non-lvalue expression whose value is the address of the object for which the function is called.

NOTE: I'm talking of constructors as functions that are "called"; there have been discussions on c.l.c++.m as to whether constructors are "functions" and to whether this terminology is correct or not; I think it is both intuitive and in agreement with the standard wording.

Steve Adamczyk: See also issue 397, which is defining a new syntax term for the body of a function including the ctor-initializers.

Notes from the March 2004 meeting:

This will be resolved when issue 397 is resolved.

Proposed resolution (October, 2005):

1. Change 11.4 [dcl.fct.def] paragraph 1 as indicated:

> Function definitions have the form
>
> *function-definition:*
> *decl-specifier-seq*$_{opt}$ *declarator* ~~*ctor-initializer*$_{opt}$~~ *function-body*
> ~~*decl-specifier-seq*$_{opt}$ *declarator function-try-block*~~
> *function-body:*
> ***ctor-initializer*$_{opt}$** *compound-statement*
> ***function-try-block***
>
> **An informal reference to the body of a function should be interpreted as a reference to the nonterminal *function-body*.**

2. Change the definition of *function-try-block* in 18 [except] paragraph 1:

> *function-try-block:*
> `try` *ctor-initializer*$_{opt}$ ~~*function-body*~~ ***compound-statement*** *handler-seq*

3. Change 6.3.7 [basic.scope.class] paragraph 1, point 1, as indicated:

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

4. Change 6.3.7 [basic.scope.class] paragraph 1, point 5, as indicated:

> The potential scope of a declaration that extends to or past the end of a class definition also extends to the regions defined by its member definitions, even if the members are defined lexically outside the class (this includes static data member definitions, nested class definitions, member function definitions (including the member function body ~~and, for constructor functions (15.1 [class.ctor]), the ctor-initializer (15.6.2 [class.base.init])~~) and any portion of the declarator part of such definitions which follows the identifier, including a *parameter-declaration-clause* and any default arguments (11.3.6 [dcl.fct.default]). [*Example:*...

5. Change footnote 32 in 6.4.1 [basic.lookup.unqual] paragraph 8 as indicated:

> That is, an unqualified name that occurs, for instance, in a type or default argument expression in the ~~*parameter-declaration-clause*~~, ***parameter-declaration-clause*** **or** in the function body~~, or in an expression of a *mem-initializer* in a constructor definition~~.

6. Change _N4567_.5.1.1 [expr.prim.general] paragraph 3 as indicated:

> ...The keyword `this` shall be used only inside a non-static class member function body (12.2.1 [class.mfct]) ~~or in a constructor *mem-initializer* (15.6.2 [class.base.init])~~...

7. Change 12.2 [class.mem] paragraph 2 as indicated:

> ...Within the class *member-specification*, the class is regarded as complete within function bodies, default arguments, **and** *exception-specifications*~~, and constructor *ctor-initializers*~~ (including such things in nested classes)...

8. Change 12.2 [class.mem] paragraph 9 as indicated:

> Each occurrence in an expression of the name of a non-static data member or non-static member function of a class shall be expressed as a class member access (8.2.5 [expr.ref]), except when it appears in the formation of a pointer to member (8.3.1 [expr.unary.op])~~, or~~ **or** when it appears in the body of a non-static member function of its class or of a class derived from its class (12.2.2 [class.mfct.non-static])~~, or when it appears in a *mem-initializer* for a constructor for its class or for a class derived from its class (15.6.2 [class.base.init])~~.

9. Change the note in 12.2.1 [class.mfct] paragraph 5 as indicated:

> [*Note:* a name used in a member function definition (that is, in the *parameter-declaration-clause* including the default arguments (11.3.6 [dcl.fct.default])~~, or~~ **or** in the member function body~~, or, for a constructor function (15.1 [class.ctor]), in a mem-initializer expression (15.6.2 [class.base.init])~~) is looked up as described in 6.4 [basic.lookup]. — *end note*]

10. Change 12.2.2 [class.mfct.non-static] paragraph 1 as indicated:

    ...A non-static member function may also be called directly using the function call syntax (8.2.2 [expr.call], 16.3.1.1 [over.match.call]) **from within the body of a member function of its class or of a class derived from its class.**

    - ~~from within the body of a member function of its class or of a class derived from its class, or~~
    - ~~from a *mem-initializer* (15.6.2 [class.base.init]) for a constructor for its class or for a class derived from its class.~~

11. Change 12.2.2 [class.mfct.non-static] paragraph 3 as indicated:

    When an *id-expression* (_N4567_.5.1.1 [expr.prim.general]) that is not part of a class member access syntax (8.2.5 [expr.ref]) and not used to form a pointer to member (8.3.1 [expr.unary.op]) is used in the body of a non-static member function of class `X` ~~or used in the *mem-initializer* for a constructor of class X~~, if name lookup (6.4.1 [basic.lookup.unqual]) resolves the name in the *id-expression* to a non-static non-type member of class `X` or of a base class of `X`, the *id-expression* is transformed into a class member access expression (8.2.5 [expr.ref]) using `(*this)` (12.2.2.1 [class.this]) as the *postfix-expression* to the left of the `.` operator...

12. Change 15.1 [class.ctor] paragraph 7 as indicated:

    ...The implicitly-defined default constructor performs the set of initializations of the class that would be performed by a user-written default constructor for that class with ~~an empty *mem-initializer-list*~~ **no *ctor-initializer*** (15.6.2 [class.base.init]) and an empty ~~function body~~ ***compound-statement***...

13. Change 15.6.2 [class.base.init] paragraph 4 as indicated:

    ...After the call to a constructor for class `X` has completed, if a member of `X` is neither specified in the constructor's *mem-initializer*s, nor default-initialized, nor value-initialized, nor given a value during execution of **the *compound-statement* of** the body of the constructor, the member has indeterminate value.

14. Change the last bullet of 15.6.2 [class.base.init] paragraph 5 as indicated:

    - Finally, the ~~body~~ ***compound-statement*** of the constructor **body** is executed.

15. Change 18 [except] paragraph 4 as indicated:

    A *function-try-block* associates a *handler-seq* with the *ctor-initializer*, if present, and the ~~*function-body*~~ ***compound-statement***. An exception thrown during the execution of the initializer expressions in the *ctor-initializer* or during the execution of the ~~*function-body*~~ ***compound-statement*** transfers control to a handler in a *function-try-block* in the same way as an exception thrown during the execution of a *try-block* transfers control to other handlers. [*Example:*

    ```
    int f(int);
    class C {
        int i;
        double d;
    public:
        C(int, double);
    };

    C::C(int ii, double id)
    try
        : i(f(ii)), d(id)
    {
        // constructor ~~function-body~~ statements
    }
    catch (...)
    {
        // handles exceptions thrown from the ctor-initializer
        // and from the constructor ~~function-body~~ statements
    }
    ```

    —*end example*]

16. Change 18.2 [except.ctor] paragraph 2 as indicated:

    When an exception is thrown, control is transferred to the nearest handler with a matching type (18.3 [except.handle]); "nearest" means the handler for which the ~~*compound-statement*~~, ***compound-statement*** **or** *ctor-initializer*~~, or *function-body*~~ following the `try` keyword was most recently entered by the thread of control and not yet exited.

---

406. Static data member in class with name for linkage purposes

Section: 12.2.3.2 [class.static.data] **Status:** CD1 **Submitter:** Jorgen Bundgaard **Date:** 12 Mar 2003

[Voted into WP at March 2004 meeting.]

The following test program is claimed to be a negative C++ test case for "Unnamed classes shall not contain static data members", c.f. ISO/IEC 14882 section 12.2.3.2 [class.static.data] paragraph 5.

```
struct B {
      typedef struct {
            static int i;          // Is this legal C++ ?
      } A;
```

```
};

int B::A::i = 47;      // Is this legal C++ ?
```

We are not quite sure about what an "unnamed class" is. There is no exact definition in ISO/IEC 14882; the closest we can come to a hint is the wording of section 10.1.3 [dcl.typedef] paragraph 5, where it seems to be understood that a class-specifier with no identifier between "class" and "{" is unnamed. The identifier provided after "}" ("A" in the test case above) is there for "linkage purposes" only.

To us, class B::A in the test program above seems "named" enough, and there is certainly a mechanism to provide the definition for B::A::i (in contrast to the note in section 12.2.3.2 [class.static.data] paragraph 5).

Our position is therefore that the above test program is indeed legal C++. Can you confirm or reject this claim?

Herb Sutter replied to the submitter as follows: Here are my notes based on a grep for "unnamed class" in the standard:

- 6.5 [basic.link] paragraph 4, bullet 3, makes a note that does not directly speak to your question but which draws the same distinction:

  > a named class (clause class), or an unnamed class defined in a typedef declaration in which the class has the typedef name for linkage purposes (10.1.3 [dcl.typedef]);

  Likewise in your example, you have an unnamed class defined in a typedef declaration.
- 12.2.3.2 [class.static.data] paragraph 5 does indeed appear to me to make your example not supported by ISO C++ (although implementations could allow it as an extension, and many implementation do happen to allow it).
- 10.1.3 [dcl.typedef] paragraph 5 does indeed likewise confirm the interpretation you give of an unnamed class.

So yes, an unnamed class is one where there is no identifier (class name) between the class-key and the {. This is also in harmony with the production for class-name in 12 [class] paragraph 1:

> *class-name*:
> *identifier*
> *template-id*

Notes from the October 2003 meeting:

We agree that the example is not valid; this is an unnamed class. We will add wording to define an unnamed class. The note in 12.2.3.2 [class.static.data] paragraph 5 should be corrected or deleted.

Proposed Resolution (October 2003):

At the end of clause 12 [class], paragraph 1, add the following:

> A *class-specifier* where the *class-head* omits the optional *identifier* defines an unnamed class.

Delete the following from 12.2.3.2 [class.static.data] paragraph 5:

> [*Note:* this is because there is no mechanism to provide the definitions for such static data members.]

---

454. When is a definition of a static data member required?

Section: 12.2.3.2 [class.static.data] **Status:** CD1 **Submitter:** Gennaro Prota **Date:** 18 Jan 2004

[Voted into WP at the October, 2006 meeting.]

As a result of the resolution of core issue 48, the current C++ standard is not in sync with existing practice and with user expectations as far as definitions of static data members having const integral or const enumeration type are concerned. Basically what current implementations do is to require a definition only if the address of the constant is taken. Example:

```
void f() {

  std::string s;
  ...

  // current implementations don't require a definition
  if (s.find('a', 3) == std::string::npos) {
   ...
  }
```

To the letter of the standard, though, the above requires a definition of `npos`, since the expression `std::string::npos` is potentially evaluated. I think this problem would be easily solved with simple changes to 12.2.3.2 [class.static.data] paragraph 4, 12.2.3.2 [class.static.data] paragraph 5 and 6.2 [basic.def.odr] paragraph 3.

Suggested resolution:

Replace 12.2.3.2 [class.static.data] paragraph 4 with:

> If a static data member is of const integral or const enumeration type, its declaration in the class definition can specify a constant-initializer which shall be [note1] an integral constant expression (5.19). In that case, the member can appear in integral constant expressions. No definition of the member is required, unless an lvalue expression that designates it is potentially evaluated and either used as operand to the built-in unary & operator [note 2] or directly bound to a reference.

If a definition exists, it shall be at namespace scope and shall not contain an initializer.

In 12.2.3.2 [class.static.data] paragraph 5 change

There shall be exactly one definition of a static data member that is used in a program; no diagnostic is required; see 3.2.

to

Except as allowed by 9.4.2 par. 4, there shall be exactly one definition of a static data member that is potentially evaluated (3.2) in a program; no diagnostic is required.

In 6.2 [basic.def.odr] paragraph 3 add, at the beginning:

Except for the omission allowed by 9.4.2, par. 4, ...

[note 1] Actually it shall be a "= followed by a constant-expression". This could probably be an editorial fix, rather than a separate DR.

[note 2] Note that this is the case when reinterpret_cast-ing to a reference, like in

```
struct X { static const int value = 0; };
const char & c = reinterpret_cast<const char&>(X::value);
```

See 8.2.10 [expr.reinterpret.cast]/10

More information, in response to a question about why issue 48 does not resolve the problem:

The problem is that the issue was settled in a way that solves much less than it was supposed to solve; that's why I decided to file, so to speak, a DR on a DR.

I understand this may seem a little 'audacious' on my part, but please keep reading. Quoting from the text of DR 48 (emphasis mine):

Originally, all static data members still had to be defined outside the class whether they were used or not.

But that restriction was supposed to be lifted [...]

In particular, if an integral/enum const static data member is initialized within the class, *and its address is never taken*, we agreed that no namespace-scope definition was required.

The corresponding resolution doesn't reflect this intent, with the definition being still required in most situations anyway: it's enough that the constant appears outside a place where constants are *required* (ignoring the obvious cases of sizeof and typeid) and you have to provide a definition. For instance:

```
struct X {
 static const int c = 1;
};

void f(int n)
{
 if (n == X::c)   // <-- potentially evaluated
  ...
}
```

<start digression>

Most usages of non-enum BOOST_STATIC_COSTANTs, for instance, are (or were, last time I checked) non-conforming. If you recall, Paul Mensonides pointed out that the following template

```
// map_integral

template<class T, T V> struct map_integral : identity<T> {
  static const T value = V;
};

template<class T, T V> const T map_integral<T, V>::value;
```

whose main goal is to map the same couples (type, value) to the same storage, also solves the definition problem. In this usage it is an excellent hack (if your compiler is good enough), but IMHO still a hack on a language defect.

<end digression>

What I propose is to solve the issue according to the original intent, which is also what users expect and all compilers that I know of already do. Or, in practice, we would have a rule that exists only as words in a standard document.

PS: I've sent a copy of this to Mr. Adamczyk to clarify an important doubt that occurred to me while writing this reply:

if no definition is provided for an integral static const data member is that member an object? Paragraph 1.8/1 seems to say no, and in fact it's difficult to think it is an object without assuming/pretending that a region of storage exists for it (an object *is* a region of storage according to the standard).

I would think that when no definition is required we have to assume that it could be a non-object. In that case there's nothing in 3.2 which says what 'used' means for such an entity and the current wording would thus be defective. Also, since the name of the member is an lvalue and 3.10/2 says an lvalue refers to an object we would have another problem.

OTOH the standard could pretend it is always an object (though the compiler can optimize it away) and in this case it should probably make a special case for it in 3.2/2.

Notes from the March 2004 meeting:

We sort of like this proposal, but we don't feel it has very high priority. We're not going to spend time discussing it, but if we get drafting for wording we'll review it.

Proposed resolution (October, 2005):

1. Change the first two sentences of 6.2 [basic.def.odr] paragraph 2 from:

   > An expression is *potentially evaluated* unless it appears where an integral constant expression is required (see 8.20 [expr.const]), is the operand of the `sizeof` operator (8.3.3 [expr.sizeof]), or is the operand of the `typeid` operator and the expression does not designate an lvalue of polymorphic class type (8.2.8 [expr.typeid]). An object or non-overloaded function is *used* if its name appears in a potentially-evaluated expression.

   to

   > An expression that is the operand of the `sizeof` operator (8.3.3 [expr.sizeof]) is *unevaluated*, as is an expression that is the operand of the `typeid` operator if it is not an lvalue of a polymorphic class type (8.2.8 [expr.typeid]); all other expressions are *potentially evaluated*. An object or non-overloaded function whose name appears as a potentially-evaluated expression is *used*, unless it is an object that satisfies the requirements for appearing in an integral constant expression (8.20 [expr.const]) and the lvalue-to-rvalue conversion (7.1 [conv.lval]) is immediately applied.

2. Change the first sentence of 12.2.3.2 [class.static.data] paragraph 2 as indicated:

   > If a static data member is of const integral or const enumeration type, its declaration in the class definition can specify a *constant-initializer* ~~which~~ **whose *constant-expression*** shall be an integral constant expression (8.20 [expr.const]).

---

58. Signedness of bit fields of enum type

Section: 12.2.4 [class.bit] **Status:** CD1 **Submitter:** Steve Adamczyk **Date:** 13 Oct 1998

[Voted into WP at the October, 2006 meeting.]

Section 12.2.4 [class.bit] paragraph 4 needs to be more specific about the signedness of bit fields of enum type. How much leeway does an implementation have in choosing the signedness of a bit field? In particular, does the phrase "large enough to hold all the values of the enumeration" mean "the implementation decides on the signedness, and then we see whether all the values will fit in the bit field", or does it require the implementation to make the bit field signed or unsigned if that's what it takes to make it "large enough"?

(See also issue 172.)

Note (March, 2005): Clark Nelson observed that there is variation among implementations on this point.

Notes from April, 2005 meeting:

Although implementations enjoy a great deal of latitude in handling bit-fields, it was deemed more user-friendly to ensure that the example in paragraph 4 will work by requiring implementations to use an unsigned underlying type if the enumeration type has no negative values. (If the implementation is allowed to choose a signed representation for such bit-fields, the comparison against `TRUE` will be `false`.)

In addition, it was observed that there is an apparent circularity between 10.2 [dcl.enum] paragraph 7 and 12.2.4 [class.bit] paragraph 4 that should be resolved.

Proposed resolution (April, 2006):

1. Replace 10.2 [dcl.enum] paragraph 7, deleting the embedded footnote 85, with the following:

   > For an enumeration where e_{min} is the smallest enumerator and e_{max} is the largest, the values of the enumeration are the values in the range b_{min} to b_{max}, defined as follows: Let K be 1 for a two's complement representation and 0 for a one's complement or sign-magnitude representation. b_{max} is the smallest value greater than or equal to $\max(|e_{min}|-K, |e_{max}|)$ and equal to 2^M-1, where M is a non-negative integer. b_{min} is zero if e_{min} is non-negative and $-(b_{max}+K)$ otherwise. The size of the smallest bit-field large enough to hold all the values of the enumeration type is $\max(M,1)$ if b_{min} is zero and $M+1$ otherwise. It is possible to define an enumeration that has values not defined by any of its enumerators.

2. Add the indicated text to the second sentence of 12.2.4 [class.bit] paragraph 4:

   > If the value of an enumerator is stored into a bit-field of the same enumeration type and the number of bits in the bit-field is large enough to hold all the values of that enumeration type **(10.2 [dcl.enum])**, the original enumerator value and the value of the bit-field shall compare equal.

---

436. Problem in example in 9.6 paragraph 4

Section: 12.2.4 [class.bit] **Status:** CD1 **Submitter:** Roberto Santos **Date:** 10 October 2003

[Voted into WP at October 2004 meeting.]

It looks like the example on 12.2.4 [class.bit] paragraph 4 has both the enum and function contributing the identifier "f" for the same scope.

```
enum BOOL { f=0, t=1 };
struct A {
  BOOL b:1;
};
A a;
void f() {
  a.b = t;
  if (a.b == t) // shall yield true
  { /* ... */ }
}
```

Proposed resolution:

Change the example at the end of 12.2.4 [class.bit]/4 from:

```
enum BOOL { f=0, t=1 };
struct A {
  BOOL b:1;
};
A a;
void f() {
  a.b = t;
  if (a.b == t) // shall yield true
  { /* ... */ }
}
```

To:

```
enum BOOL { FALSE=0, TRUE=1 };
struct A {
  BOOL b:1;
};
A a;
void f() {
  a.b = TRUE;
  if (a.b == TRUE) // shall yield true
  { /* ... */ }
}
```

---

198. Definition of "use" in local and nested classes

Section: 12.4 [class.local] **Status:** CD1 **Submitter:** Erwin Unruh **Date:** 27 Jan 2000

[Voted into WP at April 2003 meeting.]

12.4 [class.local] paragraph 1 says,

> Declarations in a local class can use only type names, static variables, `extern` variables and functions, and enumerators from the enclosing scope.

The definition of when an object or function is "used" is found in 6.2 [basic.def.odr] paragraph 2 and essentially says that the operands of `sizeof` and non-polymorphic `typeid` operators are not used. (The resolution for issue 48 will add contexts in which integral constant expressions are required to the list of non-uses.)

This definition of "use" would presumably allow code like

```
void foo() {
    int i;
    struct S {
        int a[sizeof(i)];
    };
};
```

which is required for C compatibility.

However, the restrictions on nested classes in 12.2.5 [class.nest] paragraph 1 are very similar to those for local classes, and the example there explicitly states that a reference in a `sizeof` expression is a forbidden use (abbreviated for exposition):

```
class enclose {
public:
    int x;
    class inner {
        void f(int i)
        {
            int a = sizeof(x);  // error: refers to enclose::x
        }
    };
};
```

[As a personal note, I have seen real-world code that was exactly like this; it was hard to persuade the author that the required writearound, `sizeof(((enclose*) 0)->x)`, was an improvement over `sizeof(x)`. —*wmm*]

Similarly, 12.2 [class.mem] paragraph 9 would appear to prohibit examples like the following:

```
struct B {
    char x[10];
};
struct D: B {
    char y[sizeof(x)];
};
```

Suggested resolution: Add cross-references to 6.2 [basic.def.odr] following the word "use" in both 12.2.5 [class.nest] and 12.4 [class.local], and change the example in 12.2.5 [class.nest] to indicate that a reference in a `sizeof` expression is permitted. In 12.2 [class.mem] paragraph 9, "referred to" should be changed to "used" with a cross_reference to 6.2 [basic.def.odr].

Notes from 10/01 meeting:

It was noted that the suggested resolution did not make the `sizeof()` example in 12.2.5 [class.nest] valid. Although the reference to the argument of `sizeof()` is not regarded as a use, the right syntax must be used nonetheless to reference a non-static member from the enclosing class. The use of the member name by itself is not valid. The consensus within the core working group was that nothing should be done about this case. It was later discovered that 12.2.3 [class.static] paragraph 3 states that

> The definition of a `static` member shall not use directly the names of the nonstatic members of its class or of a base class of its class (including as operands of the `sizeof` operator). The definition of a `static` member may only refer to these members to form pointer to members (8.3.1 [expr.unary.op]) or with the class member access syntax (8.2.5 [expr.ref]).

This seems to reinforce the decision of the working group.

The use of "use" should still be cross-referenced. The statements in 12.2.5 [class.nest] and 12.4 [class.local] should also be rewritten to state the requirement positively rather than negatively as the list of "can't"s is already missing some cases such as template parameters.

Notes from the 4/02 meeting:

We backed away from "use" in the technical sense, because the requirements on the form of reference are the same whether or not the reference occurs inside a `sizeof`.

Proposed Resolution (revised October 2002):

In 12.2 [class.mem] paragraph 9, replace

> Except when used to form a pointer to member (8.3.1 [expr.unary.op]), when used in the body of a nonstatic member function of its class or of a class derived from its class (12.2.2 [class.mfct.non-static]), or when used in a *mem-initializer* for a constructor for its class or for a class derived from its class (15.6.2 [class.base.init]), a nonstatic data or function member of a class shall only be referred to with the class member access syntax (8.2.5 [expr.ref]).

with the following paragraph

> Each occurrence in an expression of the name of a nonstatic data member or nonstatic member function of a class shall be expressed as a class member access (8.2.5 [expr.ref]), except when it appears in the formation of a pointer to member (8.3.1 [expr.unary.op]), when it appears in the body of a nonstatic member function of its class or of a class derived from its class (12.2.2 [class.mfct.non-static]), or when it appears in a *mem-initializer* for a constructor for its class or for a class derived from its class (15.6.2 [class.base.init]).

In 12.2.5 [class.nest] paragraph 1, replace the last sentence,

> Except by using explicit pointers, references, and object names, declarations in a nested class can use only type names, static members, and enumerators from the enclosing class.

with the following

> [Note: In accordance with 12.2 [class.mem], except by using explicit pointers, references, and object names, declarations in a nested class shall not use nonstatic data members or nonstatic member functions from the enclosing class. This restriction applies in all constructs including the operands of the `sizeof` operator.]

In the example following 12.2.5 [class.nest] paragraph 1, change the comment on the first statement of function `f` to emphasize that `sizeof(x)` is an error. The example reads in full:

```
int x;
int y;
class enclose {
public:
  int x;
  static int s;
  class inner {
    void f(int i)
    {
      int a = sizeof(x);  // error: direct use of enclose::x even in sizeof
      x = i;              // error: assign to enclose::x
      s = i;              // OK: assign to enclose::s
      ::x = i;            // OK: assign to global x
      y = i;              // OK: assign to global y
    }
    void g(enclose* p, int i)
    {
      p->x = i;        // OK: assign to enclose::x
    }
  };
};
```

```
inner* p = 0;           // error: inner not in scope
```

---

484. Can a *base-specifier* name a cv-qualified class type?

Section: 13 [class.derived] **Status:** CD1 **Submitter:** Richard Corden **Date:** 21 Oct 2004

[Voted into WP at the October, 2006 meeting.]

Issue 298, recently approved, affirms that cv-qualified class types can be used as *nested-name-specifier*s. Should the same be true for *base-specifier*s?

Rationale (April, 2005):

The resolution of issue 298 added new text to 12.1 [class.name] paragraph 5 making it clear that a typedef that names a cv-qualified class type is a *class-name*. Because the definition of *base-specifier* simply refers to *class-name*, it is already the case that cv-qualified class types are permitted as *base-specifier*s.

Additional notes (June, 2005):

It's not completely clear what it means to have a cv-qualified type as a *base-specifier*. The original proposed resolution for issue 298 said that "the cv-qualifiers are ignored," but that wording is not in the resolution that was ultimately approved.

If the cv-qualifiers are *not* ignored, does that mean that the base-class subobject should be treated as always similarly cv-qualified, regardless of the cv-qualification of the derived-class lvalue used to access the base-class subobject? For instance:

```
typedef struct B {
    void f();
    void f() const;
    int i;
} const CB;

struct D: CB { };

void g(D* dp) {
    dp->f();    // which B::f?
    dp->i = 3;  // permitted?
}
```

Proposed resolution (October, 2005):

1. Change 12.1 [class.name] paragraph 5 as indicated:

   > A *typedef-name* (10.1.3 [dcl.typedef]) that names a class type, or a cv-qualified version thereof, is also a ~~*class-name*, but~~ ***class-name*. If a *typedef-name* that names a cv-qualified class type is used where a *class-name* is required, the cv-qualifiers are ignored. A *typedef-name*** shall not be used as the *identifier* in a *class-head*.

2. Delete 10.1.3 [dcl.typedef] paragraph 8:

   > [*Note:* if the *typedef-name* is used where a *class-name* (or *enum-name*) is required, the program is ill-formed. For example,
   >
   > ```
   > typedef struct {
   >     S();      // error: requires a return type because S is
   >               // an ordinary member function, not a constructor
   > } S;
   > ```
   >
   > —*end note*]

---

39. Conflicting ambiguity rules

Section: 13.2 [class.member.lookup] **Status:** CD1 **Submitter:** Neal M Gafter **Date:** 20 Aug 1998

[Voted into WP at April 2005 meeting.]

The ambiguity text in 13.2 [class.member.lookup] may not say what we intended. It makes the following example ill-formed:

```
struct A {
    int x(int);
};
struct B: A {
    using A::x;
    float x(float);
};

int f(B* b) {
    b->x(3);  // ambiguous
}
```

This is a name lookup ambiguity because of 13.2 [class.member.lookup] paragraph 2:

... Each of these declarations that was introduced by a using-declaration is considered to be from each sub-object of C that is of the type containing the declaration designated by the using-declaration. If the resulting set of declarations are not all from sub-objects of the same type, or the set has a nonstatic member and includes members from distinct sub-objects, there is an ambiguity and the program is ill-formed.

This contradicts the text and example in paragraph 12 of 10.3.3 [namespace.udecl] .

Proposed Resolution (10/00):

1. Replace the two cited sentences from 13.2 [class.member.lookup] paragraph 2 with the following:

   The resulting set of declarations shall all be from sub-objects of the same type, or there shall be a set of declarations from sub-objects of a single type that contains *using-declaration*s for the declarations found in all other sub-object types. Furthermore, for nonstatic members, the resulting set of declarations shall all be from a single sub-object, or there shall be a set of declarations from a single sub-object that contains *using-declarations* for the declarations found in all other sub-objects. Otherwise, there is an ambiguity and the program is ill-formed.

2. Replace the examples in 13.2 [class.member.lookup] paragraph 3 with the following:

```
struct A {
    int x(int);
    static int y(int);
};
struct V {
    int z(int);
};
struct B: A, virtual V {
    using A::x;
    float x(float);
    using A::y;
    static float y(float);
    using V::z;
    float z(float);
};
struct C: B, A, virtual V {
};

void f(C* c) {
    c->x(3);    // ambiguous -- more than one sub-object A
    c->y(3);    // not ambiguous
    c->z(3);    // not ambiguous
}
```

Notes from 04/01 meeting:

The following example should be accepted but is rejected by the wording above:

```
struct A { static void f(); };

struct B1: virtual A {
    using A::f;
};

struct B2: virtual A {
    using A::f;
};

struct C: B1, B2 { };

void g() {
    C::f();        // OK, calls A::f()
}
```

Notes from 10/01 meeting (Jason Merrill):

The example in the issues list:

```
struct A {
    int x(int);
};
struct B: A {
    using A::x;
    float x(float);
};

int f(B* b) {
    b->x(3);  // ambiguous
}
```

Is broken under the existing wording:

> ... Each of these declarations that was introduced by a using-declaration is considered to be from each sub-object of C that is of the type containing the declaration designated by the using-declaration. If the resulting set of declarations are not all from sub-objects of the same type, or the set has a nonstatic member and includes members from distinct sub-objects, there is an ambiguity and the program is ill-formed.

Since the two x's are considered to be "from" different objects, looking up x produces a set including declarations "from" different objects, and the program is ill-formed. Clearly this is wrong. The problem with the existing wording is that it fails to consider lookup context.

The first proposed solution:

> The resulting set of declarations shall all be from sub-objects of the same type, or there shall be a set of declarations from sub-objects of a single type that contains *using-declaration*s for the declarations found in all other sub-object types. Furthermore, for nonstatic members, the resulting set of declarations shall all be from a single sub-object, or there shall be a set of declarations from a single sub-object that contains *using-declarations* for the declarations found in all other sub-objects. Otherwise, there is an ambiguity and the program is ill-formed.

breaks this testcase:

```
struct A { static void f(); };

struct B1: virtual A {
    using A::f;
};

struct B2: virtual A {
    using A::f;
};

struct C: B1, B2 { };

void g() {
    C::f();        // OK, calls A::f()
}
```

because it considers the lookup context, but not the definition context; under this definition of "from", the two declarations found are the using-declarations, which are "from" B1 and B2.

The solution is to separate the notions of lookup and definition context. I have taken an algorithmic approach to describing the strategy.

Incidentally, the earlier proposal allows one base to have a superset of the declarations in another base; that was an extension, and my proposal does not do that. One algorithmic benefit of this limitation is to simplify the case of a virtual base being hidden along one arm and not another ("domination"); if we allowed supersets, we would need to remember which subobjects had which declarations, while under the following resolution we need only keep two lists, of subobjects and declarations.

Proposed resolution (October 2002):

Replace 13.2 [class.member.lookup] paragraph 2 with:

> The following steps define the result of name lookup for a member name f in a class scope C.
>
> The *lookup set* for f in C, called S(f,C), consists of two component sets: the *declaration set*, a set of members named f; and the *subobject set*, a set of subobjects where declarations of these members (possibly including using-declarations) were found. In the declaration set, using-declarations are replaced by the members they designate, and type declarations (including injected-class-names) are replaced by the types they designate. S(f,C) is calculated as follows.
>
> If C contains a declaration of the name f, the declaration set contains every declaration of f in C (excluding bases), the subobject set contains C itself, and calculation is complete.
>
> Otherwise, S(f,C) is initially empty. If C has base classes, calculate the lookup set for f in each direct base class subjobject B_i, and merge each such lookup set S(f,B_i) in turn into S(f,C).
>
> The following steps define the result of merging lookup set S(f,B_i) into the intermediate S(f,C):
>
> - If each of the subobject members of S(f,B_i) is a base class subobject of at least one of the subobject members of S(f,C), S(f,C) is unchanged and the merge is complete. Conversely, if each of the subobject members of S(f,C) is a base class subobject of at least one of the subobject members of S(f,B_i), the new S(f,C) is a copy of S(f,B_i).
> - Otherwise, if the declaration sets of S(f,B_i) and S(f,C) differ, the merge is ambiguous: the new S(f,C) is a lookup set with an invalid declaration set and the union of the subobject sets. In subsequent merges, an invalid declaration set is considered different from any other.
> - Otherwise, consider each declaration d in the set, where d is a member of class A. If d is a nonstatic member, compare the A base class subobjects of the subobject members of S(f,B_i) and S(f,C). If they do not match, the merge is ambiguous, as in the previous step. [*Note:* It is not necessary to remember which A subobject each member comes from, since using-declarations don't disambiguate.]
> - Otherwise, the new S(f,C) is a lookup set with the shared set of declarations and the union of the subobject sets.
>
> The result of name lookup for f in C is the declaration set of S(f,C). If it is an invalid set, the program is ill-formed.
>
> [*Example:*
>
> ```
> struct A { int x; };                    // S(x,A) = {{ A::x }, { A }}
> struct B { float x; };                  // S(x,B) = {{ B::x }, { B }}
> struct C: public A, public B { };       // S(x,C) = { invalid, { A in C, B in C }}
> struct D: public virtual C { };         // S(x,D) = S(x,C)
> struct E: public virtual C { char x; }; // S(x,E) = {{ E::x }, { E }}
> struct F: public D, public E { };       // S(x,F) = S(x,E)
>
> int main() {
>   F f;
>   f.x = 0;   // OK, lookup finds { E::x }
> }
> ```
>
> S(x,F) is unambiguous because the A and B base subobjects of D are also base subobjects of E, so S(x,D) is discarded in the first merge step. --*end example*]

Turn 13.2 [class.member.lookup] paragraphs 5 and 6 into notes.

Notes from October 2003 meeting:

Mike Miller raised some new issues in N1543, and we adjusted the proposed resolution as indicated in that paper.

Further information from Mike Miller (January 2004):

Unfortunately, I've become aware of a minor glitch in the proposed resolution for issue 39 in N1543, so I'd like to suggest a change that we can discuss in Sydney.

A brief review and background of the problem: the major change we agreed on in Kona was to remove detection of multiple-subobject ambiguity from class lookup (13.2 [class.member.lookup]) and instead handle it as part of the class member access expression. It was pointed out in Kona that 14.2 [class.access.base]/5 has this effect:

> If a class member access operator, including an implicit "this->," is used to access a nonstatic data member or nonstatic member function, the reference is ill-formed if the left operand (considered as a pointer in the "." operator case) cannot be implicitly converted to a pointer to the naming class of the right operand.

After the meeting, however, I realized that this requirement is not sufficient to handle all the cases. Consider, for instance,

```
struct B {
    int i;
};

struct I1: B { };
struct I2: B { };

struct D: I1, I2 {
    void f() {
        i = 0;    // not ill-formed per 11.2p5
    }
};
```

Here, both the object expression ("this") and the naming class are "D", so the reference to "i" satisfies the requirement in 14.2 [class.access.base]/5, even though it involves a multiple-subobject ambiguity.

In order to address this problem, I proposed in N1543 to add a paragraph following 8.2.5 [expr.ref]/4:

> If E2 is a non-static data member or a non-static member function, the program is ill-formed if the class of E1 cannot be unambiguously converted (10.2) to the class of which E2 is directly a member.

That's not quite right. It does diagnose the case above as written; however, it breaks the case where qualification is used to circumvent the ambiguity:

```
struct D2: I1, I2 {
    void f() {
        I2::i = 0;    // ill-formed per proposal
    }
};
```

In my proposed wording, the class of "this" can't be converted to "B" (the qualifier is ignored), so the access is ill-formed. Oops.

I think the following is a correct formulation, so the proposed resolution we discuss in Sydney should contain the following paragraph instead of the one in N1543:

> If E2 is a nonstatic data member or a non-static member function, the program is ill-formed if **the naming class (11.2) of E2** cannot be unambiguously converted (10.2) to the class of which E2 is directly a member.

This reformulation also has the advantage of pointing readers to 14.2 [class.access.base], where the the convertibility requirement from the class of E1 to the naming class is located and which might otherwise be overlooked.

Notes from the March 2004 meeting:

We discussed this further and agreed with these latest recommendations. Mike Miller has produced a paper N1626 that gives just the final collected set of changes.

(This resolution also resolves isssue 306.)

---

306. Ambiguity by class name injection

Section: 13.2 [class.member.lookup] **Status:** CD1 **Submitter:** Clark Nelson **Date:** 19 Jul 2001

[Voted into WP at April 2005 meeting.]

Is the following well-formed?

```
struct A {
    struct B { };
};
struct C : public A, public A::B {
    B *p;
};
```

The lookup of `B` finds both the struct `B` in `A` and the injected `B` from the `A::B` base class. Are they the same thing? Does the standard say so?

What if a struct is found along one path and a typedef to that struct is found along another path? That should probably be valid, but does the standard say so?

This is resolved by issue 39

February 2004: Moved back to "Review" status because issue 39 was moved back to "Review".

---

390. Pure virtual must be defined when implicitly called

Section: 13.4 [class.abstract] **Status:** CD1 **Submitter:** Daniel Frey **Date:** 14 Nov 2002

[Voted into WP at March 2004 meeting.]

In clause 13.4 [class.abstract] paragraph 2, it reads:

> A pure virtual function need be defined only if explicitly called with the qualified-id syntax (_N4567_.5.1.1 [expr.prim.general]).

This is IMHO incomplete. A dtor is a function (well, a "special member function", but this also makes it a function, right?) but it is called implicitly and thus without a qualified-id syntax. Another alternative is that the pure virtual function is called directly or indirectly from the ctor. Thus the above sentence which specifies when a pure virtual function need be defined ("...only if...") needs to be extended:

> A pure virtual function need be defined only if explicitly called with the qualified-id syntax (_N4567_.5.1.1 [expr.prim.general]) or if implicitly called (15.4 [class.dtor] or 15.7 [class.cdtor]).

Proposed resolution:

Change 13.4 [class.abstract] paragraph 2 from

> A pure virtual function need be defined only if explicitly called with the *qualified-id* syntax (_N4567_.5.1.1 [expr.prim.general]).

to

> A pure virtual function need be defined only if ~~explicitly~~ called with**, or as if with (15.4 [class.dtor]),** the *qualified-id* syntax (_N4567_.5.1.1 [expr.prim.general]).

Note: 15.4 [class.dtor] paragraph 6 defines the "as if" cited.

---

8. Access to template arguments used in a function return type and in the nested name specifier

Section: 14 [class.access] **Status:** CD1 **Submitter:** Mike Ball **Date:** unknown

[Moved to DR at 4/01 meeting.]

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 14 [class.access] paragraph 5 we have:

> All access controls in clause 11 affect the ability to access a class member name from a particular scope... 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 declarator.

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.

Rationale:

Not a defect. This behavior is as intended.

ISSUE 2:

Now consider `void A::int_temp<A::A1, A::garbconst + 2, &A::func>::func1() {...}` By my reading of 14.7 [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?

Proposed resolution (04/01):

The resolution for this issue is contained in the resolution for issue 45.

---

494. Problems with the resolution of issue 45

Section: 14 [class.access] **Status:** CD1 **Submitter:** Lloyd J. Lewins **Date:** 17 Dec 2004

[Voted into WP at the October, 2006 meeting.]

The proposed resolution for issue 45 inserts the following sentence after 14 [class.access] paragraph 1:

> A member of a class can also access all names as the class of which it is a member.

I don't think that this is correctly constructed English. I see two possibilities:

1. This is a typo, and the correct change is:

   > A member of a class can also access all names of the class of which it is a member.

2. The intent is something more like:

   > A member of a nested class can also access all names accessible by any other member of the class of which it is a member.

[Note: this was editorially corrected at the time defect resolutions were being incorporated into the Working Paper to read, "...can also access all the names declared in the class of which it is a member," which is essentially the same as the preceding option 1.]

I would prefer to use the language proposed for 14.7 [class.access.nest]:

> A nested class is a member and as such has the same access rights as any other member.

A second problem is with the text in 14.3 [class.friend] paragraph 2:

> [*Note:* this means that access to private and protected names is also granted to member functions of the friend class (as if the functions were each friends) and to the static data member definitions of the friend class. This also means that private and protected type names from the class granting friendship can be used in the *base-clause* of a nested class of the friend class. However, the declarations of members of classes nested within the friend class cannot access the names of private and protected members from the class granting friendship. Also, because the *base-clause* of the friend class is not part of its member declarations, the *base-clause* of the friend class cannot access the names of the private and protected members from the class granting friendship. For example,
>
> ```
> class A {
>     class B { };
>     friend class X;
> };
> class X : A::B {      // ill-formed: A::B cannot be accessed
>                       // in the base-clause for X
>     A::B mx;          // OK: A::B used to declare member of X
>     class Y: A::B {   // OK: A::B used to declare member of X
>         A::B my;      // ill-formed: A::B cannot be accessed
>                       // to declare members of nested class of X
>     };
> };
> ```
>
> —*end note*]

This seems to be an oversight. The proposed change to 14.7 [class.access.nest] paragraph 1 would appear to have eliminated the restrictions on nested class access. However, at least one compiler (gcc 3.4.3) doesn't appear to take my view, and continues with the restrictions on access by classes within a friend class, while implementing the rest of the resolution of issue 45.

Note (March, 2005):

Andreas Hommel: I think issue 45 requires an additional change in 12.2.5 [class.nest] paragraph 4:

> Like a member function, a friend function (14.3 [class.friend]) defined within a nested class is in the lexical scope of that class; it obeys the same rules for name binding as a static member function of that class (12.2.3 [class.static]) and has no special access rights to members of an enclosing class.

I believe the "no special access rights" language should be removed.

Proposed resolution (October, 2005):

This issue is resolved by the resolution of issue 372.

---

9. Clarification of access to base class members

Section: 14.2 [class.access.base] **Status:** CD1 **Submitter:** unknown **Date:** unknown

[Moved to DR at 4/01 meeting.]

14.2 [class.access.base] paragraph 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*`.

(See J16/99-0042 = WG21 N1218.)

Proposed Resolution (04/01):

1. Add preceding 14.2 [class.access.base] paragraph 4:

   A base class `B` of `N` is *accessible* at *R*, if
   - an invented public member of `B` would be a public member of `N`, or
   - *R* occurs in a member or friend of class `N`, and an invented public member of `B` would be a private or protected member of `N`, or
   - *R* occurs in a member or friend of a class `P` derived from `N`, and an invented public member of `B` would be a private or protected member of `P`, or
   - there exists a class `S` such that `B` is a base class of `S` accessible at *R* and `S` is a base class of `N` accessible at *R*. [*Example:*

   ```
   class B {
   public:
       int m;
   };

   class S: private B {
       friend class N;
   };

   class N: private S {
       void f() {
           B* p = this;  // OK because class S satisfies the
                         // fourth condition above: B is a base
                         // class of N accessible in f() because
                         // B is an accessible base class of S
                         // and S is an accessible base class of N.
       }
   };
   ```

   —*end example*]

2. Delete the first sentence of 14.2 [class.access.base] paragraph 4:

   > A base class is said to be accessible if an invented public member of the base class is accessible.

3. Replace the last sentence ("A member *m* is accessible...") by the following:

A member `m` is accessible at the point *R* when named in class `N` if
- `m` as a member of `N` is public, or
- `m` as a member of `N` is private, and *R* occurs in a member or friend of class `N`, or
- `m` as a member of `N` is protected, and *R* occurs in a member or friend of class `N`, or in a member or friend of a class `P` derived from `N`, where `m` as a member of `P` is private or protected, or
- there exists a base class `B` of `N` that is accessible at *R*, and `m` is accessible at *R* when named in class `B`. [*Example:*...

The resolution for issue 207 modifies this wording slightly.

---

16. Access to members of indirect private base classes

Section: 14.2 [class.access.base] **Status:** CD1 **Submitter:** unknown **Date:** unknown

[Moved to DR at 4/01 meeting.]

The text in 14.2 [class.access.base] paragraph 4 does not seem to handle the following cases:

```
class D;

class B {
private:
    int i;
    friend class D;
};

class C : private B { };

class D : private C {
    void f() {
        B::i; //1: well-formed?
        i;    //2: well-formed?
    }
};
```

The member `i` is not a member of `D` and cannot be accessed in the scope of `D`. What is the naming class of the member `i` on line `//1` and line `//2`?

Proposed Resolution (04/01): The resolution for this issue is contained in the resolution for issue 9..

---

207. *using-declaration*s and protected access

Section: 14.2 [class.access.base] **Status:** CD1 **Submitter:** Jason Merrill **Date:** 28 Feb 2000

[Moved to DR at 10/01 meeting.]

Consider the following example:

```
class A {
protected:
  static void f() {};
};

class B : A {
public:
  using A::f;
  void g() {
    A::f();
  }
};
```

The standard says in 14.2 [class.access.base] paragraph 4 that the call to `A::f` is ill-formed:

> A member *m* is accessible when named in class *N* if
>
> - *m* as a member of *N* is public, or
> - *m* as a member of *N* is private, and the reference occurs in a member or friend of class *N*, or
> - *m* as a member of *N* is protected, and the reference occurs in a member or friend of class *N*, or in a member or friend of a class *P* derived from *N*, where *m* as a member of *P* is private or protected, or
> - there exists a base class *B* of *N* that is accessible at the point of reference, and *m* is accessible when named in class *B*.

Here, *m* is `A::f` and *N* is `A`.

- `f` as a member of `A` is public? **No**.
- `f` as a member of `A` is private? **No**.
- `f` as a member of `A` is protected? **Yes**.
  - reference in a member or friend of `A`? **No**.
  - reference in a member or friend of a class derived from `A`? **Yes**, `B`.
    - `f` as a member of `B` private or protected? **No**, public.
- base of `A` accessible at point of reference? **No**.

It seems clear to me that the third bullet should say "public, private or protected".

Steve Adamczyk:The words were written before *using-declaration*s existed, and therefore didn't anticipate this case.

Proposed resolution (04/01):

Modify the third bullet of the third change ("A member *m* is accessible...") in the resolution of issue 9 to read "public, private, or protected" instead of "private or protected."

---

77. The definition of friend does not allow nested classes to be friends

Section: 14.3 [class.friend] **Status:** CD1 **Submitter:** Judy Ward **Date:** 15 Dec 1998

[Moved to DR at 4/02 meeting.]

The definition of "friend" in 14.3 [class.friend] says:

> A friend of a class is a function or class that is not a member of the class but is permitted to use the private and protected member names from the class. ...

A nested class, i.e. INNER in the example below, is a member of class OUTER. The sentence above states that it cannot be a friend. I think this is a mistake.

```
class OUTER {
    class INNER;
    friend class INNER;
    class INNER {};
};
```

Proposed resolution (04/01):

Change the first sentence of 14.3 [class.friend] as follows:

> A friend of a class is a function or class that is ~~not a member of the class but is allowed~~ **given permission** to use the private and protected member names from the class. ~~The name of a friend is not in the scope of the class, and the friend is not called with the member access operators (8.2.5 [expr.ref]) unless it is a member of another class.~~ **A class specifies its friends, if any, by way of friend declarations. Such declarations give special access rights to the friends, but they do not make the nominated friends members of the befriending class.**

---

500. Access in *base-specifier*s of friend and nested classes

Section: 14.3 [class.friend] **Status:** CD1 **Submitter:** Andreas Hommel **Date:** 25 Jan 2005

[Voted into WP at the October, 2006 meeting.]

I don't know the reason for this distinction, but it seems to be surprising that `Base::A` is legal and `D` is illegal in this example:

```
class D;
class Base
{
    class A;
    class B;
    friend class D;
};
class Base::B
{
};
class Base::A : public Base::B  // OK because of issue 45
{
};
class D : public Base::B        // illegal because of 11.4p4
{
};
```

Shouldn't this be consistent (either way)?

Notes from the April, 2005 meeting:

In discussing issue 372, the CWG decided that access in the *base-specifier*s of a class should be the same as for its members, and that resolution will apply to `friend` declarations, as well.

Proposed resolution (October, 2005):

This issue is resolved by the resolution of issue 372.

---

385. How does protected member check of 11.5 interact with using-declarations?

Section: 14.4 [class.protected] **Status:** CD1 **Submitter:** Vincent Korstanje **Date:** 24 Sep 2002

[Voted into WP at October 2004 meeting.]

We consider it not unreasonable to do the following

```
class A {
  protected:
  void g();
};
class B : public A {
  public:
    using A::g; // B::g is a public synonym for A::g
};

class C: public A {
  void foo();
};

void C::foo() {
  B b;
  b.g();
}
```

However the EDG front-end does not like and gives the error

```
#410-D: protected function "A::g" is not accessible through a "B" pointer or  object
  b.g();
    ^
```

Steve Adamczyk: The error in this case is due to 14.4 [class.protected] of the standard, which is an additional check on top of the other access checking. When that section says "a protected nonstatic member function ... of a base class" it doesn't indicate whether the fact that there is a using-declaration is relevant. I'd say the current wording taken at face value would suggest that the error is correct -- the function is protected, even if the using-declaration for it makes it accessible as a public function. But I'm quite sure the wording in 14.4 [class.protected] was written before using-declarations were invented and has not been reviewed since for consistency with that addition.

Notes from April 2003 meeting:

We agreed that the example should be allowed.

Proposed resolution (April 2003, revised October 2003):

Change 14.4 [class.protected] paragraph 1 from

> When a friend or a member function of a derived class references a protected nonstatic member function or protected nonstatic data member of a base class, an access check applies in addition to those described earlier in clause 14 [class.access]. [Footnote: This additional check does not apply to other members, *e.g.* static data members or enumerator member constants.] Except when forming a pointer to member (8.3.1 [expr.unary.op]), the access must be through a pointer to, reference to, or object of the derived class itself (or any class derived from that class (8.2.5 [expr.ref]). If the access is to form a pointer to member, the *nested-name-specifier* shall name the derived class (or any class derived from that class).

to

> An additional access check beyond those described earlier in clause 14 [class.access] is applied when a nonstatic data member or nonstatic member function is a protected member of its naming class (14.2 [class.access.base]). [Footnote: This additional check does not apply to other members, e.g., static data members or enumerator member constants.] As described earlier, access to a protected member is granted because the reference occurs in a friend or member of some class `C`. If the access is to form a pointer to member (8.3.1 [expr.unary.op]), the *nested-name-specifier* shall name `C` or a class derived from `C`. All other accesses involve a (possibly implicit) object expression (8.2.5 [expr.ref]). In this case, the class of the object expression shall be `C` or a class derived from `C`.

Additional discussion (September, 2004):

Steve Adamczyk: I wonder if this wording is incorrect. Consider:

```
class A {
public:
  int p;
};
class B : protected A {
  // p is a protected member of B
};
class C : public B {
  friend void fr();
};
void fr() {
  B *pb = new B;
  pb->p = 1;  // Access okay?  Naming class is B, p is a protected member of B,
              // the "C" of the issue 385 wording is C, but access is not via
              // an object of type C or a derived class thereof.
}
```

I think the formulation that the member is a protected member of its naming class is not what we want. I think we intended that the member is protected in the declaration that is found, where the declaration found might be a *using-declaration*.

Mike Miller: I think the proposed wording makes the access `pb->p` ill-formed, and I think that's the right thing to do.

First, protected inheritance of `A` by `B` means that `B` intends the public and protected members of `A` to be part of `B`'s implementation, available to `B`'s descendants only. (That's why there's a restriction on converting from `B*` to `A*`, to enforce `B`'s intention on the use of members of `A`.) Consequently, I see no difference in access policy between your example and

```
class B {
protected:
    int p;
};
```

Second, the reason we have this rule is that `C`'s use of inherited protected members might be different from their use in a sibling class, say `D`. Thus members and friends of `C` can only use `B::p` in a manner consistent with `C`'s usage, i.e., in `C` or derived-from-`C` objects. If we rewrote your example slightly,

```
class D: public B { };

void fr(B* pb) {
    pb->p = 1;
}

void g() {
    fr(new D);
}
```

it's clear that the intent of this rule is broken — `fr` would be accessing `B::p` assuming `C`'s policies when the object in question actually required `D`'s policies.

(See also issues 471 and 472.)

---

10. Can a nested class access its own class name as a qualified name if it is a private member of the enclosing class?

Section: 14.7 [class.access.nest] **Status:** CD1 **Submitter:** Josee Lajoie **Date:** unknown

[Moved to DR at 4/01 meeting.]

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

This prevents 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; // ill-formed
```

This is because the nested class does not have access to the member `E` in `D`. 14 [class.access] 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`. 14 [class.access] paragraph 6 covers the access for `D::E::m`, but it doesn't affect the `D::E` access. Are there any implementations that are standard compliant that support 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 14 [class.access] paragraph 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.

Proposed resolution (04/01):

The resolution for this issue is incorporated into the resolution for issue 45.

---

45. Access to nested classes

Section: 14.7 [class.access.nest] **Status:** CD1 **Submitter:** Daveed Vandevoorde **Date:** 29 Sep 1998

[Moved to DR at 4/01 meeting.]

Example:

```
#include <iostream.h>

class C {  // entire body is private
    struct Parent {
        Parent() { cout << "C::Parent::Parent()\n"; }
    };

    struct Derived : Parent {
        Derived() { cout << "C::Derived::Derived()\n"; }
    };

    Derived d;
};


int main() {
    C c;      //  Prints message from both nested classes
    return 0;
}
```

How legal/illegal is this? Paragraphs that seem to apply here are:

14 [class.access] paragraph 1:

> A member of a class can be
>
> - `private`; that is, its name can be used only by members and friends of the class in which it is declared. [...]

and 14.7 [class.access.nest] paragraph 1:

> The members of a nested class have no special access to members of an enclosing class, nor to classes or functions that have granted friendship to an enclosing class; the usual access rules (clause 14 [class.access]) shall be obeyed. [...]

This makes me think that the '`: Parent`' part is OK by itself, but that the implicit call of '`Parent::Parent()`' by '`Derived::Derived()`' is not.

From Mike Miller:

I think it is completely legal, by the reasoning given in the (non-normative) 14.7 [class.access.nest] paragraph 2. The use of a private nested class as a base of another nested class is explicitly declared to be acceptable there. I think the rationale in the comments in the example ("// OK because of injection of name A in A") presupposes that public members of the base class will be public members in a (publicly-derived) derived class, regardless of the access of the base class, so the constructor invocation should be okay as well.

I can't find anything normative that explicitly says that, though.

(See also papers J16/99-0009 = WG21 N1186, J16/00-0031 = WG21 N1254, and J16/00-0045 = WG21 N1268.)

Proposed Resolution (04/01):

1. Insert the following as a new paragraph following 14 [class.access] paragraph 1:

   > A member of a class can also access all names as the class of which it is a member. A local class of a member function may access the same names that the member function itself may access. [*Footnote:* Access permissions are thus transitive and cumulative to nested and local classes.]

2. Delete 14 [class.access] paragraph 6.

3. In 14.7 [class.access.nest] paragraph 1, change

   > The members of a nested class have no special access to members of an enclosing class, nor to classes or functions that have granted friendship to an enclosing class; the usual access rules (clause 14 [class.access]) shall be obeyed.

   to

   > A nested class is a member and as such has the same access rights as any other member.

   Change

   ```
   B b;       // error: E::B is private
   ```

   to

   ```
   B b;       // Okay, E::I can access E::B
   ```

   Change

   ```
   p->x = i;      // error: E::x is private
   ```

   to

   ```
   p->x = i;      // Okay, E::I can access E::x
   ```

4. Delete 14.7 [class.access.nest] paragraph 2.

(This resolution also resolves issues 8 and 10.

---

263. Can a constructor be declared a friend?

Section: 15.1 [class.ctor] **Status:** CD1 **Submitter:** Martin Sebor **Date:** 13 Nov 2000

[Voted into WP at April 2003 meeting.]

According to 15.1 [class.ctor] paragraph 1, a declaration of a constructor has a special limited syntax, in which only *function-specifier*s are allowed. A `friend` specifier is not a *function-specifier*, so one interpretation is that a constructor cannot be declared in a `friend` declaration.

(It should also be noted, however, that neither `friend` nor *function-specifier* is part of the **declarator** syntax, so it's not clear that anything conclusive can be derived from the wording of 15.1 [class.ctor].)

Notes from 04/01 meeting:

The consensus of the core language working group was that it should be permitted to declare constructors as `friend`s.

Proposed Resolution (revised October 2002):

Change paragraph 1a in 6.4.3.1 [class.qual] (added by the resolution of issue 147) as follows:

> If the *nested-name-specifier* nominates a class `C`, and the name specified after the *nested-name-specifier*, when looked up in `C`, is the injected-class-name of `C` (clause 12 [class]), the name is instead considered to name the constructor of class `C`. Such a constructor name shall be used only in the *declarator-id* of a ~~constructor definition~~ **declaration** that ~~appears outside of the class definition~~ **names a constructor**....

Note: the above does not allow qualified names to be used for in-class declarations; see 11.3 [dcl.meaning] paragraph 1. Also note that issue 318 updates the same paragraph.

Change the example in 14.3 [class.friend], paragraph 4 as follows:

```
class Y {
  friend char* X::foo(int);
  friend X::X(char);    // constructors can be friends
  friend X::~X();       // destructors can be friends
  //...
};
```

---

326. Wording for definition of trivial constructor

Section: 15.1 [class.ctor] **Status:** CD1 **Submitter:** James Kanze **Date:** 9 Dec 2001

[Voted into WP at October 2003 meeting.]

In 15.1 [class.ctor] paragraph 5, the standard says "A constructor is trivial if [...]", and goes on to define a trivial default constructor. Taken literally, this would mean that a copy constructor can't be trivial (contrary to 15.8 [class.copy] paragraph 6). I suggest changing this to "A default constructor is trivial if [...]". (I think the change is purely editorial.)

Proposed Resolution (revised October 2002):

Change 15.1 [class.ctor] paragraph 5-6 as follows:

> A *default* constructor for a class `X` is a constructor of class `X` that can be called without an argument. If there is no ~~*user-declared*~~ **user-declared** constructor for class `X`, a default constructor is implicitly declared. An ~~*implicitly-declared*~~ **implicitly-declared** default constructor is an `inline public` member of its class. A **default** constructor is *trivial* if it is ~~an~~ implicitly-declared ~~default constructor~~ and if:
>
> - its class has no virtual functions (13.3 [class.virtual]) and no virtual base classes (13.1 [class.mi]), and
> - all the direct base classes of its class have trivial **default** constructors, and
> - for all the nonstatic data members of its class that are of class type (or array thereof), each such class has a trivial **default** constructor.
>
> Otherwise, the **default** constructor is non-trivial.

Change 15.4 [class.dtor] paragraphs 3-4 as follows (the main changes are removing italics):

> If a class has no ~~*user-declared*~~ **user-declared** destructor, a destructor is declared implicitly. An ~~*implicitly-declared*~~ **implicitly-declared** destructor is an `inline public` member of its class. A destructor is *trivial* if it is ~~an~~ implicitly-declared ~~destructor~~ and if:
>
> - all of the direct base classes of its class have trivial destructors and
> - for all of the non-static data members of its class that are of class type (or array thereof), each such class has a trivial destructor.
>
> Otherwise, the destructor is ~~*non-trivial*~~ **non-trivial**.

In 12.3 [class.union] paragraph 1, change "trivial constructor" to "trivial default constructor".

In 15.2 [class.temporary] paragraph 3, add to the reference to 15.1 [class.ctor] a second reference, to 15.8 [class.copy].

---

331. Allowed copy constructor signatures

Section: 15.1 [class.ctor] **Status:** CD1 **Submitter:** Richard Smith **Date:** 8 Jan 2002

[Voted into WP at October 2003 meeting.]

15.1 [class.ctor] paragraph 10 states

> A copy constructor for a class X is a constructor with a first parameter of type X & or of type const X &. [Note: see 15.8 [class.copy] for more information on copy constructors.]

No mention is made of constructors with first parameters of types volatile X & or const volatile X &. This statement seems to be in contradiction with 15.8 [class.copy] paragraph 2 which states

> A non-template constructor for class X is a copy constructor if its first parameter is of type X &, const X &, volatile X & or const volatile X &, ...

15.8 [class.copy] paragraph 5 also mentions the volatile versions of the copy constructor, and the comparable paragraphs for copy assignment (15.8 [class.copy] paragraphs 9 and 10) all allow volatile versions, so it seems that 15.1 [class.ctor] is at fault.

Proposed resolution (October 2002):

Change 15.1 [class.ctor] paragraph 10 from

> A *copy constructor* for a class `X` is a constructor with a first parameter of type `X&` or of type `const X&`. [*Note:* see 15.8 [class.copy] for more information on copy constructors.]

to (note that the dropping of italics is intentional):

> A copy constructor (15.8 [class.copy]) is used to copy objects of class type.

---

86. Lifetime of temporaries in query expressions

Section: 15.2 [class.temporary] **Status:** CD1 **Submitter:** Steve Adamczyk **Date:** Jan 1999

[Voted into WP at April, 2006 meeting.]

In 15.2 [class.temporary] paragraph 5, should binding a reference to the result of a "`?`" operation, each of whose branches is a temporary, extend both temporaries?

Here's an example:

```
const SFileName &C = noDir ? SFileName("abc") : SFileName("bcd");
```

Do the temporaries created by the `SFileName` conversions survive the end of the full expression?

Notes from 10/00 meeting:

Other problematic examples include cases where the temporary from one branch is a base class of the temporary from the other (i.e., where the implementation must remember which type of temporary must be destroyed), or where one branch is a temporary and the other is not. Similar questions also apply to the comma operator. The sense of the core language working group was that implementations should be required to support these kinds of code.

Notes from the March 2004 meeting:

We decided that the cleanest model is one in which any "?" operation that returns a class rvalue always copies one of its operands to a temporary and returns the temporary as the result of the operation. (Note that this may involve slicing.) An implementation would be free to optimize this using the rules in 15.8 [class.copy] paragraph 15, and in fact we would expect that in many cases compilers would do such optimizations. For example, the compiler could construct both rvalues in the above example into a single temporary, and thus avoid a copy.

See also issue 446.

Proposed resolution (October, 2004):

This issue is resolved by the resolutions of issue 446.

Note (October, 2005):

This issue was overlooked when issue 446 was moved to "ready" status and was thus inadvertently omitted from the list of issues accepted as Defect Reports at the October, 2005 meeting.

---

124. Lifetime of temporaries in default initialization of class arrays

Section: 15.2 [class.temporary] **Status:** CD1 **Submitter:** Jack Rouse **Date:** 3 June 1999

[Moved to DR at 4/01 meeting.]

Jack Rouse: 15.2 [class.temporary] states that temporary objects will normally be destroyed at the end of the full expression in which they are created. This can create some unique code generation requirements when initializing a class array with a default constructor that uses a default argument. Consider the code:

```
struct T {
   int i;
   T( int );
   ~T();
};

struct S {
   S( int = T(0).i );
   ~S();
};

S* f( int n )
{
   return new S[n];
}
```

The full expression allocating the array in `f(int)` includes the default constructor for `S`. Therefore according to 4.6 [intro.execution] paragraph 14, it includes the default argument expression for `S(int)`. So evaluation of the full expression should include evaluating the default argument "n" times and creating "n" temporaries of type `T`. But the destruction of the temporaries must be delayed until the end of the full expression so this requires allocating space at runtime for "n" distinct temporaries. It is unclear how these temporaries are supposed to be allocated and deallocated. They cannot readily be autos because a variable allocation is required.

I believe that many existing implementations will destroy the temporaries needed by the default constructor after each array element is initialized. But I can't find anything in the standard that allows the temporaries to be destroyed early in this case.

I think the standard should allow the early destruction of temporaries used in the default initialization of class array elements. I believe early destruction is the status quo, and I don't think the users of existing C++ compilers have been adversely impacted by it.

Proposed resolution (04/01):

The proposed resolution is contained in the proposal for issue 201.

---

199. Order of destruction of temporaries

Section: 15.2 [class.temporary] **Status:** CD1 **Submitter:** Alan Nash **Date:** 27 Jan 2000

[Voted into the WP at the April, 2007 meeting as part of paper J16/07-0099 = WG21 N2239.]

15.2 [class.temporary] paragraph 3 simply states the requirement that temporaries created during the evaluation of an expression

> are destroyed as the last step in evaluating the full-expression (1.9) that (lexically) contains the point where they were created.

There is nothing said about the relative order in which these temporaries are destroyed.

Paragraph 5, dealing with temporaries bound to references, says

> the temporaries created during the evaluation of the expression initializing the reference, except the temporary to which the reference is bound, are destroyed at the end of the full-expression in which they are created and in the reverse order of the completion of their construction.

Is this difference intentional? May temporaries in expressions other than those initializing references be deleted in non-LIFO order?

Notes from 04/00 meeting:

Steve Adamczyk expressed concern about constraining implementations that are capable of fine-grained parallelism -- they may be unable to determine the order of construction without adding undesirable overhead.

Proposed resolution (April, 2007):

As specified in paper J16/07-0099 = WG21 N2239.

---

201. Order of destruction of temporaries in initializers

Section: 15.2 [class.temporary] **Status:** CD1 **Submitter:** Alan Nash **Date:** 31 Jan 2000

[Moved to DR at 4/01 meeting.]

According to 15.2 [class.temporary] paragraph 4, an expression appearing as the initializer in an object definition constitutes a context "in which temporaries are destroyed at a different point than the end of the full-expression." It goes on to say that the temporary containing the value of the expression persists until after the initialization is complete (see also issue 117). This seems to presume that the end of the full-expression is a point earlier than the completion of the initialization.

However, according to 4.6 [intro.execution] paragraphs 12-13, the full-expression in such cases is, in fact, the entire initialization. If this is the case, the behavior described for temporaries in an initializer expression is simply the normal behavior of temporaries in any expression, and treating it as an exception to the general rule is both incorrect and confusing.

Proposed resolution (04/01):

[Note: this proposal also addresses issue 124.]

1. Add to the end of 4.6 [intro.execution] paragraph 12:

   > If the initializer for an object or sub-object is a full-expression, the initialization of the object or sub-object (e.g., by calling a constructor or copying an expression value) is considered to be part of the full-expression.

2. Replace 15.2 [class.temporary] paragraph 4 with:

   > There are two contexts in which temporaries are destroyed at a different point than the end of the full-expression. The first context is when a default constructor is called to initialize an element of an array. If the constructor has one or more default arguments, any temporaries created in the default argument expressions are destroyed immediately after return from the constructor.

---

320. Question on copy constructor elision example

Section: 15.2 [class.temporary] **Status:** CD1 **Submitter:** Steve Clamage **Date:** 2 Nov 2001

[Voted into WP at April 2005 meeting.]

Section 15.2 [class.temporary] paragraph 2, abridged:

```
X f(X);
void g()
{
      X a;
      a = f(a);
}
```

`a=f(a)` requires a temporary for either the argument `a` or the result of `f(a)` to avoid undesired aliasing of `a`.

The note seems to imply that an implementation is allowed to omit copying "a" to f's formal argument, or to omit using a temporary for the return value of f. I don't find that license in normative text.

Function f returns an X by value, and in the expression the value is assigned (not copy-constructed) to "a". I don't see how that temporary can be omitted. (See also 15.8 [class.copy] p 15)

Since "a" is an lvalue and not a temporary, I don't see how copying "a" to f's formal parameter can be avoided.

Am I missing something, or is 15.2 [class.temporary] p 2 misleading?

Proposed resolution (October, 2004):

In 15.2 [class.temporary] paragraph 2, change the last sentence as indicated:

> On the other hand, the expression `a=f(a)` requires a temporary for ~~either the argument `a` or the result of `f(a)` to avoid undesired aliasing of `a`~~ **the result of `f(a)`, which is then assigned to** `a`.

---

392. Use of full expression lvalue before temporary destruction

Section: 15.2 [class.temporary] **Status:** CD1 **Submitter:** Stephen Clamage **Date:** 21 Nov 2002

[Voted into WP at March 2004 meeting.]

```
class C {
public:
    C();
    ~C();
    int& get() { return p; } // reference return
private:
    int p;
};
```

```
int main ()
{
    if ( C().get() ) // OK?
}
```

Section 15.2 [class.temporary] paragraph 3 says a temp is destroyed as the last step in evaluating the full expression. But the expression C().get() has a reference type. Does 15.2 [class.temporary] paragraph 3 require that the dereference to get a boolean result occur before the destructor runs, making the code valid? Or does the code have undefined behavior?

Bill Gibbons: It has undefined behavior, though clearly this wasn't intended. The lvalue-to-rvalue conversion that occurs in the "if" statement is not currently part of the full-expression.

From section 15.2 [class.temporary] paragraph 3:

> Temporary objects are destroyed as the last step in evaluating the full-expression (4.6 [intro.execution]) that (lexically) contains the point where they were created.

From section 4.6 [intro.execution] paragraph 12:

> A full-expression is an expression that is not a subexpression of another expression. If a language construct is defined to produce an implicit call of a function, a use of the language construct is considered to be an expression for the purposes of this definition.

The note in section 4.6 [intro.execution] paragraph 12 goes on to explain that this covers expressions used as initializers, but it does not discuss lvalues within temporaries.

It is a small point but it is probably worth correcting 4.6 [intro.execution] paragraph 12. Instead of the "implicit call of a function" wording, it might be better to just say that a full-expression includes any implicit use of the expression value in the enclosing language construct, and include a note giving implicit calls and lvalue-to-rvalue conversions as examples.

Offhand the places where this matters include: initialization (including member initializers), selection statements, iteration statements, return, throw

Proposed resolution (April 2003):

Change 4.6 [intro.execution] paragraph 12-13 to read:

> A *full-expression* is an expression that is not a subexpression of another expression. If a language construct is defined to produce an implicit call of a function, a use of the language construct is considered to be an expression for the purposes of this definition. **Conversions applied to the result of an expression in order to satisfy the requirements of the language construct in which the expression appears are also considered to be part of the full-expression.**
>
> ~~[*Note:* certain contexts in C++ cause the evaluation of a full-expression that results from a syntactic construct other than *expression* (8.19 [expr.comma]). For example, in 11.6 [dcl.init] one syntax for *initializer* is~~
>
> ~~*(expression-list)*~~
>
> ~~but the resulting construct is a function call upon a constructor function with *expression-list* as an argument list; such a function call is a full-expression. For example, in 11.6 [dcl.init], another syntax for *initializer* is~~
>
> ~~*= initializer-clause*~~
>
> ~~but again the resulting construct might be a function call upon a constructor function with one *assignment-expression* as an argument; again, the function call is a full-expression.~~] **[*Example:***
>
> ```
> struct S {
>     S(int i): I(i) { }
>     int& v() { return I; }
>   private:
>     int I;
> };
>
> S s1(1);           // full-expression is call of S::S(int)
> S s2 = 2;          // full-expression is call of S::S(int)
>
> void f() {
>     if (S(3).v())  // full-expression includes lvalue-to-rvalue and
>                    // int to bool conversions, performed before
>                    // temporary is deleted at end of full-expression
>     { }
> }
> ```
>
> ***—end example*]**

---

443. Wording nit in description of lifetime of temporaries

Section: 15.2 [class.temporary] **Status:** CD1 **Submitter:** Matthias Hofmann **Date:** 2 Dec 2003

[Voted into WP at April 2005 meeting.]

There seems to be a typo in 15.2 [class.temporary]/5, which says "The temporary to which the reference is bound or the temporary that is the complete object TO a subobject OF which the TEMPORARY is bound persists for the lifetime of the reference except as specified below."

I think this should be "The temporary to which the reference is bound or the temporary that is the complete object OF a subobject TO which the REFERENCE is bound persists for the lifetime of the reference except as specified below."

I used upper-case letters for the parts I think need to be changed.

Proposed resolution (October, 2004):

Change 15.2 [class.temporary] paragraph 5 as indicated:

> The temporary to which the reference is bound or the temporary that is the complete object ~~to~~ **of** a subobject ~~of~~ **to** which the ~~temporary~~ **reference** is bound persists for the lifetime of the reference except as specified below.

---

464. Wording nit on lifetime of temporaries to which references are bound

Section: 15.2 [class.temporary] **Status:** CD1 **Submitter:** Allan Odgaard **Date:** 21 Feb 2004

[Voted into WP at April, 2006 meeting.]

Section 15.2 [class.temporary] paragraph 5 ends with this "rule":

[...] if obj2 is an object with static or automatic storage duration created after the temporary is created, the temporary shall be destroyed after obj2 is destroyed.

For the temporary to be destroyed after obj2 is destroyed, when obj2 has static storage, I would say that the reference to the temporary should also have static storage, but that is IMHO not clear from the paragraph.

Example:

```
void f ()
{
   const T1& ref = T1();
   static T2 obj2;
   ...
}
```

Here the temporary would be destoyed *before* obj2, contrary to the rule above.

Steve Adamczyk: I agree there's a minor issue here. I think the clause quoted above meant for obj1 and obj2 to have the same storage duration. Replacing "obj2 is an object with static or automatic storage duration" by "obj2 is an object with the same storage duration as obj1" would, I believe, fix the problem.

Notes from October 2004 meeting:

We agreed with Steve Adamczyk's suggestion.

Proposed resolution (October, 2005):

Change 15.2 [class.temporary] paragraph 5 as follows:

> ... In addition, the destruction of temporaries bound to references shall take into account the ordering of destruction of objects with static or automatic storage duration (6.7.1 [basic.stc.static], 6.7.3 [basic.stc.auto]); that is, if `obj1` is an object ~~with static or automatic storage duration~~ created before the temporary is created **with the same storage duration as the temporary**, the temporary shall be destroyed before `obj1` is destroyed; if `obj2` is an object ~~with static or automatic storage duration~~ created after the temporary is created **with the same storage duration as the temporary**, the temporary shall be destroyed after `obj2` is destroyed...

---

296. Can conversion functions be static?

Section: 15.3.2 [class.conv.fct] **Status:** CD1 **Submitter:** Scott Meyers **Date:** 5 Jul 2001

[Moved to DR at October 2002 meeting.]

May user-defined conversion functions be static? That is, should this compile?

```
class Widget {
public:
  static operator bool() { return true; }
};
```

All my compilers hate it. I hate it, too. However, I don't see anything in 15.3.2 [class.conv.fct] that makes it illegal. Is this a prohibition that arises from the grammar, i.e., the grammar doesn't allow "static" to be followed by a *conversion-function-id* in a member function declaration? Or am I just overlooking something obvious that forbids static conversion functions?

Proposed Resolution (4/02):

Add to 15.3.2 [class.conv.fct] as a new paragraph 7:

> Conversion functions cannot be declared `static`.

---

244. Destructor lookup

Section: 15.4 [class.dtor] **Status:** CD1 **Submitter:** John Spicer **Date:** 6 Sep 2000

[Moved to DR at October 2002 meeting.]

15.4 [class.dtor] contains this example:

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

D D_object;
typedef B B_alias;
B* B_ptr = &D_object;

void f() {
    D_object.B::~B();              // calls B's destructor
    B_ptr->~B();                   // calls D's destructor
    B_ptr->~B_alias();             // calls D's destructor
    B_ptr->B_alias::~B();          // calls B's destructor
    B_ptr->B_alias::~B_alias();    // error, no B_alias in class B
}
```

On the other hand, 6.4.3 [basic.lookup.qual] contains this example:

```
struct C {
    typedef int I;
};
typedef int I1, I2;
extern int* p;
extern int* q;
p->C::I::~I();       // I is looked up in the scope of C
q->I1::~I2();        // I2 is looked up in the scope of
                     // the postfix-expression
struct A {
    ~A();
};
typedef A AB;
int main()
{
    AB *p;
    p->AB::~AB();    // explicitly calls the destructor for A
}
```

Note that

```
B_ptr->B_alias::~B_alias();
```

is claimed to be an error, while the equivalent

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

is claimed to be well-formed.

I believe that clause 3 is correct and that clause 12 is in error. We worked hard to get the destructor lookup rules in clause 3 to be right, and I think we failed to notice that a change was also needed in clause 12.

Mike Miller:

Unfortunately, I don't believe 6.4.3 [basic.lookup.qual] covers the case of `p->AB::~AB()`. It's clearly intended to do so, as evidenced by 6.4.3.1 [class.qual] paragraph 1 ("a destructor name is looked up as specified in 6.4.3 [basic.lookup.qual]"), but I don't think the language there does so.

The relevant paragraph is 6.4.3 [basic.lookup.qual] paragraph 5. (None of the other paragraphs in that section deal with this topic at all.) It has two parts. The first is

> If a *pseudo-destructor-name* (8.2.4 [expr.pseudo]) contains a *nested-name-specifier*, the *type-name*s are looked up as types in the scope designated by the *nested-name-specifier*.

This sentence doesn't apply, because `~AB` isn't a *pseudo-destructor-name*. 8.2.4 [expr.pseudo] makes clear that this syntactic production (8.2 [expr.post] paragraph 1) only applies to cases where the *type-name* is not a *class-name*. `p->AB::~AB` is covered by the production using *id-expression*.

The second part of 6.4.3 [basic.lookup.qual] paragraph 5 says

> In a *qualified-id* of the form:

::$_{opt}$ *nested-name-specifier* ~ *class-name*

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

::$_{opt}$ *nested-name-specifier class-name* :: ~ *class-name*

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

This wording doesn't apply, either. The first one doesn't because the *nested-name-specifier* is a *class-name*, not a namespace name. The second doesn't because there's only one layer of qualification.

As far as I can tell, there's no normative text that specifies how the `~AB` is looked up in `p->AB::~AB()`. 6.4.3.1 [class.qual], where all the other class member qualified lookups are handled, defers to 6.4.3 [basic.lookup.qual], and 6.4.3 [basic.lookup.qual] doesn't cover the case.

See also issue 305.

Jason Merrill: My thoughts on the subject were that the name we use in a destructor call is really meaningless; as soon as we see the ~ we know what the user means, all we're doing from that point is testing their ability to name the destructor in a conformant way. I think that everyone will agree that

```
anything::B::~B()
```

should be well-formed, regardless of the origins of the name "B". I believe that the rule about looking up the second "B" in the same context as the first was intended to provide this behavior, but to me this seems much more heavyweight than necessary. We don't need a whole new type of lookup to be able to use the same name before and after the ~; we can just say that if the two names match, the call is well-formed. This is significantly simpler to express, both in the standard and in an implementation.

Anyone writing two different names here is either deliberately writing obfuscated code, trying to call the destructor of a nested class, or fighting an ornery compiler (i.e. one that still wants to see `B_alias::~B()`). I think we can ignore the first case. The third would be handled by reverting to the old rule (look up the name after ~ in the normal way) with the lexical matching exception described above -- or we could decide to break such code, do no lookup at all, and only accept a matching name. In a good implementation, the second should probably get an error message telling them to write `Outer::Inner::~Inner` instead.

We discussed this at the meetings, but I don't remember if we came to any sort of consensus on a direction. I see three options:

1. Stick with the status quo, i.e. the special lookup rule such that if the name before `::~` is a class name, the name after `::~` is looked up in the same scope as the previous one. If we choose this option, we just need better wording that actually expresses this, as suggested in the issue list. This option breaks old `B_alias::~B` code where `B_alias` is declared in a different scope from `B`.
2. Revert to the old rules, whereby the name after `::~` is looked up just like a name after `::`, with the exception that if it matches the name before `::~` then it is considered to name the same class. This option supports old code and code that writes `B_alias::~B_alias`. It does not support the `q->I1::~I2` usage of 6.4.3 [basic.lookup.qual], but that seems like deliberate obfuscation. This option is simpler to implement than #1.
3. Do no lookup for a name after `::~`; it must match the name before. This breaks old code as #1, but supports the most important case where the names match. This option may be slightly simpler to implement than #2. It is certainly easier to teach.

My order of preference is 2, 3, 1.

Incidentally, it seems to me oddly inconsistent to allow *Namespace*::~*Class*, but not *Outer*::~*Inner*. Prohibiting the latter makes sense from the standpoint of avoiding ambiguity, but what was the rationale for allowing the former?

John Spicer: I agree that allowing *Namespace*::~*Class* is odd. I'm not sure where this came from. If we eliminated that special case, then I believe the #1 rule would just be that in `A::B1::~B2` you look up `B1` and `B2` in the same place in all cases.

I don't like #2. I don't think the "old" rules represent a deliberate design choice, just an error in the way the lookup was described. The usage that rule permits `p->X::~Y` (where `Y` is a typedef to `X` defined in `X`), but I doubt people really do that. In other words, I think that #1 a more useful special case than #2 does, not that I think either special case is very important.

One problem with the name matching rule is handling cases like:

```
A<int> *aip;

aip->A<int>::~A<int>();  // should work
aip->A<int>::~A<char>(); // should not
```

I would favor #1, while eliminating the special case of *Namespace*::~*Class*.

Proposed resolution (10/01):

Replace the normative text of 6.4.3 [basic.lookup.qual] paragraph 5 after the first sentence with:

Similarly, in a *qualified-id* of the form:

::$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name* :: ~ *class-name*

the second *class-name* is looked up in the same scope as the first.

In 15.4 [class.dtor] paragraph 12, change the example to

```
D D_object;
typedef B B_alias;
B* B_ptr = &D_object;

void f() {
  D_object.B::~B();               //  calls  B's destructor
```

```
  B_ptr->~B();                  //  calls  D's destructor
  B_ptr->~B_alias();            //  calls  D's destructor
  B_ptr->B_alias::~B();         //  calls  B's destructor
  B_ptr->B_alias::~B_alias();   //  calls  B's destructor
}
```

April 2003: See issue 399.

---

252. Looking up deallocation functions in virtual destructors

Section: 15.4 [class.dtor] **Status:** CD1 **Submitter:** Steve Clamage **Date:** 19 Oct 2000

[Moved to DR at 10/01 meeting.]

There is a mismatch between 15.4 [class.dtor] paragraph 11 and 15.5 [class.free] paragraph 4 regarding the lookup of deallocation functions in virtual destructors. 15.4 [class.dtor] says,

> At the point of definition of a virtual destructor (including an implicit definition (15.8 [class.copy])), non-placement operator delete shall be looked up in the scope of the destructor's class (6.4.1 [basic.lookup.unqual]) and if found shall be accessible and unambiguous. [*Note*: this assures that an operator delete corresponding to the dynamic type of an object is available for the *delete-expression* (15.5 [class.free]).]

The salient features to note from this description are:

1. The lookup is "in the scope of the destructor's class," which implies that only members are found (cf 15.2 [class.temporary]). (The cross-reference would indicate otherwise, however, since it refers to the description of looking up unqualified names; this kind of lookup "spills over" into the surrounding scope.)
2. Only non-placement operator delete is looked up. Presumably this means that a placement operator delete is ignored in the lookup.

On the other hand, 15.5 [class.free] says,

> If a *delete-expression* begins with a unary `::` operator, the deallocation function's name is looked up in global scope. Otherwise, if the *delete-expression* is used to deallocate a class object whose static type has a virtual destructor, the deallocation function is the one found by the lookup in the definition of the dynamic type's virtual destructor (15.4 [class.dtor]). Otherwise, if the *delete-expression* is used to deallocate an object of class `T` or array thereof, the static and dynamic types of the object shall be identical and the deallocation function's name is looked up in the scope of `T`. If this lookup fails to find the name, the name is looked up in the global scope. If the result of the lookup is ambiguous or inaccessible, or if the lookup selects a placement deallocation function, the program is ill-formed.

Points of interest in this description include:

1. For a class type with a virtual destructor, the lookup is described as being "in the definition of the dynamic type's virtual destructor," rather than "in the scope of the dynamic type." That is, the lookup is assumed to be an unqualified lookup, presumably terminating in the global scope.
2. The assumption is made that the lookup in the virtual destructor was successful ("...the one found...", not "...the one found..., **if any**"). This will not be the case if the deallocation function was not declared as a member somewhere in the inheritance hierarchy.
3. The lookup in the non-virtual-destructor case **does** find placement deallocation functions and can fail as a result.

Suggested resolution: Change the description of the lookup in 15.4 [class.dtor] paragraph 11 to match the one in 15.5 [class.free] paragraph 4.

Proposed resolution (10/00):

1. Replace 15.4 [class.dtor] paragraph 11 with the following:

   > At the point of definition of a virtual destructor (including an implicit definition), the non-array deallocation function is looked up in the scope of the destructor's class (13.2 [class.member.lookup]), and, if no declaration is found, the function is looked up in the global scope. If the result of this lookup is ambiguous or inaccessible, or if the lookup selects a placement deallocation function, the program is ill-formed. [*Note:* this assures that a deallocation function corresponding to the dynamic type of an object is available for the *delete-expression* (15.5 [class.free]).]

2. In 15.5 [class.free] paragraph 4, change

   > ...the deallocation function is the one found by the lookup in the definition of the dynamic type's virtual destructor (15.4 [class.dtor]).

   to

   > ...the deallocation function is the one selected at the point of definition of the dynamic type's virtual destructor (15.4 [class.dtor]).

---

272. Explicit destructor invocation and *qualified-id*s

Section: 15.4 [class.dtor] **Status:** CD1 **Submitter:** Mike Miller **Date:** 22 Feb 2001

[Moved to DR at 10/01 meeting.]

15.4 [class.dtor] paragraph 12 contains the following note:

> an explicit destructor call must always be written using a member access operator (8.2.5 [expr.ref]); in particular, the *unary-expression* `~X()` in a member function is not an explicit destructor call (8.3.1 [expr.unary.op]).

This note is incorrect, as an explicit destructor call can be written as a *qualified-id*, e.g., `X::~X()`, which does not use a member access operator.

Proposed resolution (04/01):

Change 15.4 [class.dtor] paragraph 12 as follows:

> [*Note:* an explicit destructor call must always be written using a member access operator (8.2.5 [expr.ref]) **or a *qualified-id* (_N4567_.5.1.1 [expr.prim.general])**; in particular, the *unary-expression* `~X()` in a member function is not an explicit destructor call (8.3.1 [expr.unary.op]).]

---

677. Deleted `operator delete` and virtual destructors

Section: 15.4 [class.dtor] **Status:** CD1 **Submitter:** Mike Miller **Date:** 15 February, 2008

[Voted into the WP at the September, 2008 meeting.]

Deallocation functions can't be virtual because they are static member functions; however, according to 15.5 [class.free] paragraph 7, they behave like virtual functions when the class's destructor is virtual:

> Since member allocation and deallocation functions are `static` they cannot be virtual. [*Note:* however, when the *cast-expression* of a *delete-expression* refers to an object of class type, because the deallocation function actually called is looked up in the scope of the class that is the dynamic type of the object, if the destructor is virtual, the effect is the same.

Because the intent is to make any use of a deleted function diagnosable at compile time, a virtual deleted function can neither override nor be overridden by a non-deleted function, as described in 13.3 [class.virtual] paragraph 14:

> A function with a deleted definition (11.4 [dcl.fct.def]) shall not override a function that does not have a deleted definition. Likewise, a function that does not have a deleted definition shall not override a function with a deleted definition.

One would assume that a similar kind of prohibition is needed for deallocation functions in a class hierarchy with virtual destructors, but it's not clear that the current specification says that. 11.4 [dcl.fct.def] paragraph 10 says,

> A program that refers to a deleted function implicitly or explicitly, other than to declare it, is ill-formed.

Furthermore, the deallocation function is looked up at the point of definition of a virtual destructor (15.4 [class.dtor] paragraph 11), and the function found by this lookup is considered to be "used" (6.2 [basic.def.odr] paragraph 2). However, it's not completely clear that this "use" constitutes a "reference" in the sense of 11.4 [dcl.fct.def] paragraph 10, especially in a program in which an object of a type that would call that deallocation function is never deleted.

Suggested resolution:

Augment the list of lookup results from a virtual destructor that render a program ill-formed in 15.4 [class.dtor] paragraph 10 to include a deleted function:

> If the result of this lookup is ambiguous or inaccessible, or if the lookup selects a placement deallocation function **or a function with a deleted definition (11.4 [dcl.fct.def])**, the program is ill-formed.

Proposed resolution (June, 2008):

Change 15.4 [class.dtor] paragraph 10 as follows:

> If the result of this lookup is ambiguous or inaccessible, or if the lookup selects a placement deallocation function **or a function with a deleted definition (11.4 [dcl.fct.def])**, the program is ill-formed.

---

510. Default initialization of POD classes?

Section: 15.6 [class.init] **Status:** CD1 **Submitter:** Mike Miller **Date:** 18 Mar 2005

[Voted into WP at April, 2006 meeting.]

11.6 [dcl.init] paragraph 10 makes it clear that non-static POD class objects with no initializer are left uninitialized and have an indeterminate initial value:

> If no initializer is specified for an object, and the object is of (possibly cv-qualified) non-POD class type (or array thereof), the object shall be default-initialized; if the object is of const-qualified type, the underlying class type shall have a user-declared default constructor. Otherwise, if no initializer is specified for a non-static object, the object and its subobjects, if any, have an indeterminate initial value; if the object or any of its subobjects are of const-qualified type, the program is ill-formed.

15.6 [class.init] paragraph 1, however, implies that all class objects without initializers, whether POD or not, are default-initialized:

> When no initializer is specified for an object of (possibly cv-qualified) class type (or array thereof), or the initializer has the form `()`, the object is initialized as specified in 11.6 [dcl.init]. The object is default-initialized if there is no initializer, or value-initialized if the initializer is `()`.

Proposed resolution (October, 2005):

Remove the indicated words from 15.6 [class.init] paragraph 1:

> When no initializer is specified for an object of (possibly cv-qualified) class type (or array thereof), or the initializer has the form `()`, the object is initialized as specified in 11.6 [dcl.init]. ~~The object is default-initialized if there is no initializer, or value-initialized if the initializer is `()`.~~

---

683. Requirements for trivial subobject special functions

Section: 15.8 [class.copy] **Status:** CD1 **Submitter:** Jens Maurer **Date:** 13 March, 2008

[Voted into the WP at the September, 2008 meeting (resolution in paper N2757).]

Part of the decision regarding whether a class has a trivial special function (copy constructor, copy assignment operator, default constructor) is whether its base and member subobjects have corresponding trivial member functions. However, with the advent of defaulted functions, it is now possible for a single class to have both trivial and nontrivial overloads for those functions. For example,

```
struct B {
  B(B&) = default;     // trivial
  B(const B&);         // non-trivial, because user-provided
};

struct D : B { };
```

Although `B` has *a* trivial copy constructor and thus satisfies the requirements in 15.8 [class.copy] paragraph 6, the copy constructor in `B` that would be called by the implicitly-declared copy constructor in `D` is *not* trivial. This could be fixed either by requiring that all the subobject's copy constructors (or copy assignment operators, or default constructors) be trivial or that the one that would be selected by overload resolution be trivial.

Proposed resolution (July, 2008):

Change 11.4 [dcl.fct.def] paragraph 9 as follows:

> ... A special member function that would be implicitly defined as deleted shall not be explicitly defaulted. **If a special member function for a class `X` is defaulted on its first declaration, no other special member function of the same kind (default constructor, copy constructor, or copy assignment operator) shall be declared in class `X`.** A special member function is *user-provided*...

Notes from the September, 2008 meeting:

The resolution adopted as part of paper N2757 differs from the July, 2008 proposed resolution by allowing defaulted and user-provided special member functions to coexist. Instead, a trivial class is defined as having no non-trivial copy constructors or copy assignment operators, and a trivial copy constructor or assignment operator is defined as invoking only trivial copy operations for base and member subobjects.

---

162. `(&C::f)()` with nonstatic members

Section: 16.3.1.1 [over.match.call] **Status:** CD1 **Submitter:** Steve Adamczyk **Date:** 26 Aug 1999

[Moved to DR at October 2002 meeting.]

16.3.1.1 [over.match.call] paragraph 3 says that when a call of the form

```
(&C::f)()
```

is written, the set of overloaded functions named by `C::f` must not contain any nonstatic member functions. A footnote gives the rationale: if a member of `C::f` is a nonstatic member function, `&C::f` is a pointer to member constant, and therefore the call is invalid.

This is clear, it's implementable, and it doesn't directly contradict anything else in the standard. However, I'm not sure it's consistent with some similar cases.

In 16.4 [over.over] paragraph 5, second example, it is made amply clear that when `&C::f` is used as the address of a function, e.g.,

```
int (*pf)(int) = &C::f;
```

the overload set can contain both static and nonstatic member functions. The function with the matching signature is selected, and if it is nonstatic `&C::f` is a pointer to member function, and otherwise `&C::f` is a normal pointer to function.

Similarly, 16.3.1.1.1 [over.call.func] paragraph 3 makes it clear that

```
C::f();
```

is a valid call even if the overload set contains both static and nonstatic member functions. Overload resolution is done, and if a nonstatic member function is selected, an implicit `this->` is added, if that is possible.

Those paragraphs seem to suggest the general rule that you do overload resolution first and then you interpret the construct you have according to the function selected. The fact that there are static and nonstatic functions in the overload set is irrelevant; it's only necessary that the chosen function be static or nonstatic to match the context.

Given that, I think it would be more consistent if the `(&C::f)()` case would also do overload resolution first. If a nonstatic member is chosen, the program would be ill-formed.

Proposed resolution (04/01):

1. Change the indicated text in 16.3.1.1 [over.match.call] paragraph 3:

> The fourth case arises from a *postfix-expression* of the form `&F`, where `F` names a set of overloaded functions. In the context of a function call, ~~the set of functions named by `F` shall contain only non-member functions and static member functions. [Footnote: If `F` names a non-static member function, `&F` is a pointer-to-member, which cannot be used with the function-call syntax.] And in this context using `&F` behaves the same as using~~ `&F` **is treated the same as** the name `F` by itself. Thus, `(&F)(`*expression-list*$_{\text{opt}}$`)` is simply `(F)(`*expression-list*$_{\text{opt}}$`)`, which is discussed in 16.3.1.1.1 [over.call.func]. **If the function selected by overload resolution according to 16.3.1.1.1 [over.call.func] is a nonstatic member function, the program is ill-formed. [Footnote: When `F` is a nonstatic member function, a reference of the form `&A::F` is a pointer-to-member, which cannot be used with the function-call syntax, and a reference of the form `&F` is an invalid use of the "`&`" operator on a nonstatic member function.]** (The resolution of `&F` in other contexts is described in 16.4 [over.over].)

---

239. Footnote 116 and Koenig lookup

Section: 16.3.1.1.1 [over.call.func] **Status:** CD1 **Submitter:** Steve Clamage **Date:** 2 Aug 2000

[Moved to DR at 4/01 meeting.]

In describing non-member functions in an overload set, footnote 116 (16.3.1.1.1 [over.call.func]) says,

> Because of the usual name hiding rules, these will be introduced by declarations or by *using-directive*s all found in the same block or all found at namespace scope.

At least in terms of the current state of the Standard, this is not correct: a block extern declaration does not prevent Koenig lookup from occurring. For example,

```
enum E { zero };
void f(E);
void g() {
    void f(int);
    f(zero);
}
```

In this example, the overload set will include declarations from both namespace and block scope.

(See also issue 12.)

Proposed resolution (04/01):

1. In 6.4.2 [basic.lookup.argdep] paragraph 2, change

> If the ordinary unqualified lookup of the name finds the declaration of a class member function, the associated namespaces and classes are not considered.

to

> If the ordinary unqualified lookup of the name finds the declaration of a class member function, or a block-scope function declaration that is not a *using-declaration*, the associated namespaces and classes are not considered.

and change the example to:

```
namespace NS {
    class T { };
    void f(T);
    void g(T, int);
}
NS::T parm;
void g(NS::T, float);
int main() {
```

```
    f(parm);            // OK: calls NS::f
    extern void g(NS::T, float);
    g(parm, 1);         // OK: calls g(NS::T, float)
}
```

2. In 16.3.1.1.1 [over.call.func] paragraph 3 from:

   > If the name resolves to a non-member function declaration, that function and its overloaded declarations constitute the set of candidate functions.

   to

   > If the name resolves to a set of non-member function declarations, that set of functions constitutes the set of candidate functions.

   Note that this text is also edited by issue 364. Also, remove the associated footnote 116.

---

364. Calling overloaded function with static in set, with no object

Section: 16.3.1.1.1 [over.call.func] **Status:** CD1 **Submitter:** Steve Adamczyk **Date:** 23 July 2002

[Voted into WP at October 2003 meeting.]

Consider this program:

```
struct S {
  static void f (int);
  void f (char);
};

void g () {
  S::f ('a');
}
```

G++ 3.1 rejects it, saying:

```
test.C:7: cannot call member function `void S::f(char)' without object
```

Mark Mitchell: It looks to me like G++ is correct, given 16.3.1.1.1 [over.call.func]. This case is the "unqualified function call" case described in paragraph 3 of that section. ("Unqualified" here means that there is no "x->" or "x." in front of the call, not that the name is unqualified.)

That paragraph says that you first do name lookup. It then asks you to look at what declaration is returned. (That's a bit confusing; you presumably get a set of declarations. Or maybe not; the name lookup section says that if name lookup finds a non-static member in a context like this the program is in error. But surely this program is not erroneous. Hmm.)

Anyhow, you have -- at least -- "S::f(char)" as the result of the lookup.

The keyword "this" is not in scope, so "all overloaded declarations of the function name in T become candidate functions and a contrived object of type T becomes the implied object argument." That means we get both versions of "f" at this point. Then, "the call is ill-formed, however, if overload resolution selects one of the non-static members of T in this case." Since, in this case, "S::f(char)" is the winner, the program is ill-formed.

Steve Adamczyk: This result is surprising, because we've selected a function that we cannot call, when there is another function that can be called. This should either be ambiguous, or it should select the static member function. See also 16.3.1 [over.match.funcs] paragraph 2: "Similarly, when appropriate, the context can construct an argument list that contains an implied object argument..."

Notes from October 2002 meeting:

We agreed that g++ has it right, but the standard needs to be clearer.

Proposed resolution (October 2002, revised April 2003):

Change 16.3.1.1.1 [over.call.func] paragraphs 2 and 3 as follows:

> In qualified function calls, the name to be resolved is an *id-expression* and is preceded by an `->` or `.` operator. Since the construct `A->B` is generally equivalent to `(*A).B`, the rest of clause 16 [over] assumes, without loss of generality, that all member function calls have been normalized to the form that uses an object and the `.` operator. Furthermore, clause 16 [over] assumes that the *postfix-expression* that is the left operand of the `.` operator has type "*cv* `T`" where `T` denotes a class. [Footnote: Note that cv-qualifiers on the type of objects are significant in overload resolution for both lvalue and class rvalue objects. --- end footnote] Under this assumption, the *id-expression* in the call is looked up as a member function of `T` following the rules for looking up names in classes (13.2 [class.member.lookup]). ~~If a member function is found, that function and its overloaded declarations~~ **The function declarations found by that lookup** constitute the set of candidate functions. The argument list is the *expression-list* in the call augmented by the addition of the left operand of the `.` operator in the normalized member function call as the implied object argument (16.3.1 [over.match.funcs]).
>
> In unqualified function calls, the name is not qualified by an `->` or `.` operator and has the more general form of a *primary-expression*. The name is looked up in the context of the function call following the normal rules for name lookup in function calls (~~6.4.2 [basic.lookup.argdep]~~ 6.4 [basic.lookup]). ~~If the name resolves to a non-member function declaration, that function and its overloaded declarations~~ **The function declarations found by that lookup** constitute the set of candidate functions.

~~[Footnote: Because of the usual name hiding rules, these will be introduced by declarations or by *using-directives* all found in the same block or all found at namespace scope. --- end footnote]~~ **Because of the rules for name lookup, the set of candidate functions consists (1) entirely of non-member functions or (2) entirely of member functions of some class `T`. In case (1), t**~~T~~he argument list is the same as the *expression-list* in the call. ~~If the name resolves to a nonstatic member function, then the function call is actually a member function call.~~ **In case (2), the argument list is the *expression-list* in the call augmented by the addition of an implied object argument as in a qualified function call.** If the keyword `this` (12.2.2.1 [class.this]) is in scope and refers to ~~the~~ class `T` ~~of that member function~~, or a derived class ~~thereof~~ **of** `T`, then the ~~function call is transformed into a normalized qualified function call using~~ **implied object argument is** `(*this)` ~~as the *postfix-expression* to the left of the `.` operator~~. ~~The candidate functions and argument list are as described for qualified function calls above.~~ If the keyword `this` is not in scope or refers to another class, then ~~name resolution found a static member of some class `T`. In this case, all overloaded declarations of the function name in `T` become candidate functions and~~ a contrived object of type `T` becomes the implied object argument. [Footnote: An implied object argument must be contrived to correspond to the implicit object parameter attributed to member functions during overload resolution. It is not used in the call to the selected function. Since the member functions all have the same implicit object parameter, the contrived object will not be the cause to select or reject a function. --- end footnote] **If the argument list is augmented by a contrived object and** ~~The call is ill-formed, however, if~~ overload resolution selects one of the non-static member functions of `T`**, the call is ill-formed**~~ in this case~~.

Note that issue 239 also edits paragraph 3.

---

280. Access and surrogate call functions

Section: 16.3.1.1.2 [over.call.object] **Status:** CD1 **Submitter:** Andrei Iltchenko **Date:** 16 Apr 2001

[Voted into WP at October 2003 meeting.]

According to 16.3.1.1.2 [over.call.object] paragraph 2, when the *primary-expression* `E` in the function call syntax evaluates to a class object of type "*cv* `T`", a surrogate call function corresponding to an appropriate conversion function declared in a direct or indirect base class `B` of `T` is included or not included in the set of candidate functions based on class `B` being accessible.

For instance in the following code sample, as per the paragraph in question, the expression `c(3)` calls `f2`, instead of the construct being ill-formed due to the conversion function `A::operator fp1` being inaccessible and its corresponding surrogate call function providing a better match than the surrogate call function corresponding to `C::operator fp2`:

```
void  f1(int)  {  }
void  f2(float)  {  }
typedef void  (* fp1)(int);
typedef void  (* fp2)(float);

struct  A  {
   operator fp1()
   {  return  f1;  }
};

struct  B :  private A  {  };

struct  C :  B  {
   operator  fp2()
   {  return  f2;  }
};

int  main()
{
   C  c;
   c(3);  // f2 is called, instead of the construct being ill-formed.
   return  0;
}
```

The fact that the accessibility of a base class influences the overload resolution process contradicts the fundamental language rule (6.4 [basic.lookup] paragraph 1, and 16.3 [over.match] paragraph 2) that access checks are applied only once name lookup and function overload resolution (if applicable) have succeeded.

Notes from 4/02 meeting:

There was some concern about whether 13.2 [class.member.lookup] (or anything else, for that matter) actually defines "ambiguous base class". See issue 39. See also issue 156.

Notes from October 2002 meeting:

It was suggested that the ambiguity check is done as part of the call of the conversion function.

Proposed resolution (revised October 2002):

In 16.3.1.1.2 [over.call.object] paragraph 2, replace the last sentence

> Similarly, surrogate call functions are added to the set of candidate functions for each conversion function declared in an accessible base class provided the function is not hidden within `T` by another intervening declaration.

with

> Similarly, surrogate call functions are added to the set of candidate functions for each conversion function declared in a base class of `T` provided the function is not hidden within `T` by another intervening declaration.

Replace 16.3.1.1.2 [over.call.object] paragraph 3

> If such a surrogate call function is selected by overload resolution, its body, as defined above, will be executed to convert `E` to the appropriate function and then to invoke that function with the arguments of the call.

by

> If such a surrogate call function is selected by overload resolution, the corresponding conversion function will be called to convert `E` to the appropriate function pointer or reference, and the function will then be invoked with the arguments of the call. If the conversion function cannot be called (e.g., because of an ambiguity), the program is ill-formed.

---

416. Class must be complete to allow operator lookup?

Section: 16.3.1.2 [over.match.oper] **Status:** CD1 **Submitter:** Greg Comeau **Date:** 22 May 2003

[Voted into WP at October 2004 meeting.]

Normally reference semantics allow incomplete types in certain contexts, but isn't this:

```
class A;

A& operator<<(A& a, const char* msg);
void foo(A& a)
{
  a << "Hello";
}
```

required to be diagnosed because of the op<<? The reason being that the class may actually have an op<<(const char *) in it.

What is it? un- or ill-something? Diagnosable? No problem at all?

Steve Adamczyk: I don't know of any requirement in the standard that the class be complete. There is a rule that will instantiate a class template in order to be able to see whether it has any operators. But I wouldn't think one wants to outlaw the above example merely because the user might have an operator<< in the class; if he doesn't, he would not be pleased that the above is considered invalid.

Mike Miller: Hmm, interesting question. My initial reaction is that it just uses `::operator<<`; any `A::operator<<` simply won't be considered in overload resolution. I can't find anything in the Standard that would say any different.

The closest analogy to this situation, I'd guess, would be deleting a pointer to an incomplete class; 8.3.5 [expr.delete] paragraph 5 says that that's undefined behavior if the complete type has a non-trivial destructor or an operator delete. However, I tend to think that that's because it deals with storage and resource management, not just because it might have called a different function. Generally, overload resolution that goes one way when it might have gone another with more declarations in scope is considered to be not an error, cf 10.3.3 [namespace.udecl] paragraph 9, 17.7.3 [temp.nondep] paragraph 1, etc.

So my bottom line take on it would be that it's okay, it's up to the programmer to ensure that all necessary declarations are in scope for overload resolution. Worst case, it would be like the operator delete in an incomplete class -- undefined behavior, and thus not required to be diagnosed.

16.3.1.2 [over.match.oper] paragraph 3, bullet 1, says, "If T1 is a class type, the set of member candidates is the result of the qualified lookup of `T1::operator@` (16.3.1.1.1 [over.call.func])." Obviously, that lookup is not possible if T1 is incomplete. Should 16.3.1.2 [over.match.oper] paragraph 3, bullet 1, say "complete class type"? Or does the inability to perform the lookup mean that the program is ill-formed? 6.2 [basic.def.odr] paragraph 4 doesn't apply, I don't think, because you don't know whether you'll be applying a class member access operator until you know whether the operator involved is a member or not.

Notes from October 2003 meeting:

We noticed that the title of this issue did not match the body. We checked the original source and then corrected the title (so it no longer mentions templates).

We decided that this is similar to other cases like deleting a pointer to an incomplete class, and it should not be necessary to have a complete class. There is no undefined behavior.

Proposed Resolution (October 2003):

Change the first bullet of 16.3.1.2 [over.match.oper] paragraph 3 to read:

> If T1 is a **complete** class type, the set of member candidates is the result of the qualified lookup of `T1::operator@` (16.3.1.1.1 [over.call.func]); otherwise, the set of member candidates is empty.

---

60. Reference binding and valid conversion sequences

Section: 16.3.3.1.4 [over.ics.ref] **Status:** CD1 **Submitter:** Steve Adamczyk **Date:** 13 Oct 1998

[Moved to DR at October 2002 meeting.]

Does dropping a cv-qualifier on a reference binding prevent the binding as far as overload resolution is concerned? Paragraph 4 says "Other restrictions on binding a reference to a particular argument do not affect the formation of a conversion sequence." This was intended to refer to things like access checking, but some readers have taken that to mean that any aspects of reference binding not mentioned in this section do not preclude the binding.

Proposed resolution (10/01):

In 16.3.3.1.4 [over.ics.ref] paragraph 4 add the indicated text:

> Other restrictions on binding a reference to a particular argument **that are not based on the types of the reference and the argument** do not affect the formation of a standard conversion sequence, however.

---

115. Address of template-id

Section: 16.4 [over.over] **Status:** CD1 **Submitter:** John Spicer **Date:** 7 May 1999

[Voted into WP at October 2003 meeting.]

```
template <class T> void f(T);
template <class T> void g(T);
template <class T> void g(T,T);

int main()
{
    (&f<int>);
    (&g<int>);
}
```

The question is whether `&f<int>` identifies a unique function. `&g<int>` is clearly ambiguous.

16.4 [over.over] paragraph 1 says that a function template name is considered to name a set of overloaded functions. I believe it should be expanded to say that a function template name with an explicit template argument list is also considered to name a set of overloaded functions.

In the general case, you need to have a destination type in order to identify a unique function. While it is possible to permit this, I don't think it is a good idea because such code depends on there only being one template of that name that is visible.

The EDG front end issues an error on this use of "`f`". egcs 1.1.1 allows it, but the most current snapshot of egcs that I have also issues an error on it.

It has been pointed out that when dealing with nontemplates, the rules for taking the address of a single function differ from the rules for an overload set, but this asymmetry is needed for C compatibility. This need does not exist for the template case.

My feeling is that a general rule is better than a general rule plus an exception. The general rule is that you need a destination type to be sure that the operation will succeed. The exception is when there is only one template in the set and only then when you provide values for all of the template arguments.

It is true that in some cases you can provide a shorthand, but only if you encourage a fragile coding style (that will cause programs to break when additional templates are added).

I think the standard needs to specify one way or the other how this case should be handled. My recommendation would be that it is ill-formed.

Nico Josuttis: Consider the following example:

```
template <int VAL>
int add (int elem)
{
    return elem + VAL;
}

std::transform(coll.begin(), coll.end(),
               coll.begin(),
               add<10>);
```

If John's recommendation is adopted, this code will become ill-formed. I bet there will be a lot of explanation for users necessary why this fails and that they have to change `add<10>` to something like `(int (*)(int))add<10>`.

This example code is probably common practice because this use of the STL is typical and is accepted in many current implementations. I strongly urge that this issue be resolved in favor of keeping this code valid.

Bill Gibbons: I find this rather surprising. Shouldn't a *template-id* which specifies all of the template arguments be treated like a declaration-only explicit instantiation, producing a set of ordinary function declarations? And when that set happens to contain only one function, shouldn't the example code work?

(See also issue 250.)

Notes from 04/01 meeting:

The consensus of the group was that the `add` example should not be an error.

Proposed resolution (October 2002):

In 13.4 add to the end of paragraph 2:

> [*Note:* As described in 17.9.1 [temp.arg.explicit], if deduction fails and the function template name is followed by an explicit template argument list, the *template-id* is then examined to see whether it identifies a single function template specialization. If it does, the *template-id* is considered to be an lvalue for that function template specialization. The target type is not used in that determination.]

In 17.9.1 [temp.arg.explicit] paragraph 2 insert before the first example:

> In contexts where deduction is done and fails, or in contexts where deduction is not done, if a template argument list is specified and it, along with any default template arguments, identifies a single function template specialization, then the *template-id* is an lvalue for the function template specialization.

Change the first example of 17.9.1 [temp.arg.explicit] paragraph 2:

```
template<class X, class Y> X f(Y);
void g()
{
  int i = f<int>(5.6);    // Y is deduced to be double
  int j = f(5.6);         // ill-formed: X cannot be deduced
}
```

to read:

```
template<class X, class Y> X f(Y);
void g()
{
  int i = f<int>(5.6);    // Y is deduced to be double
  int j = f(5.6);         // ill-formed: X cannot be deduced
  f<void>(f<int, bool>); // Y for outer f deduced to be
                         //   int (*)(bool)
  f<void>(f<int>);        // ill-formed: f<int> does not denote a
                         //   single template function specialization
}
```

Note: This interacts with the resolution of issue 226 (default template arguments for function templates).

---

221. Must compound assignment operators be member functions?

Section: 16.5.3 [over.ass] **Status:** CD1 **Submitter:** Jim Hyslop **Date:** 3 Apr 2000

[Moved to DR at 4/01 meeting.]

Is the intent of 16.5.3 [over.ass] paragraph 1 that *all* assignment operators be non-static member functions (including `operator+=`, `operator*=`, etc.) or only simple assignment operators (`operator=`)?

Notes from 04/00 meeting:

Nearly all references to "assignment operator" in the IS mean `operator=` and not the compound assignment operators. The ARM was specific that this restriction applied only to `operator=`. If it did apply to compound assignment operators, it would be impossible to overload these operators for `bool` operands.

Proposed resolution (04/01):

1. Change the title of 8.18 [expr.ass] from "Assignment operators" to "Assignment and compound assignment operators."
2. Change the first sentence of 8.18 [expr.ass] paragraph 1 from

   > There are several assignment operators, all of which group right-to-left. All require a modifiable lvalue as their left operand, and the type of an assignment expression is that of its left operand. The result of the assignment operation is the value stored in the left operand after the assignment has taken place; the result is an lvalue.

   to

   > The assignment operator (`=`) and the compound assignment operators all group right-to-left. All require a modifiable lvalue as their left operand and return an lvalue with the type and value of the left operand after the assignment has taken place.

Additional note (10/00): Paragraphs 2-6 of 8.18 [expr.ass] should all be understood to apply to simple assignment only and not to compound assignment operators.

---

420. postfixexpression->scalar_type_dtor() inconsistent

Section: 16.5.6 [over.ref] **Status:** CD1 **Submitter:** Markus Mauhart **Date:** 8 June 2003

[Voted into WP at April, 2006 meeting.]

Lets start with the proposed solution. In 16.5.6 [over.ref], replace line ...

postfix-expression -> *id-expression*

.... with the lines ...

postfix-expression -> *template*$_{opt}$ *id-expression*
postfix-expression -> *pseudo-destructor-name*

(This then is a copy of the two lines in 8.2 [expr.post] covering "->dtor")

Alternatively remove the sentence "It implements class member access using ->" and the syntax line following.

Reasons:

Currently stdc++ is inconsistent when handling expressions of the form "postfixexpression->scalar_type_dtor()": If "postfixexpression" is a pointer to the scalar type, it is OK, but if "postfixexpression" refers to any smart pointer class (e.g. iterator or allocator::pointer) with class specific CLASS::operator->() returning pointer to the scalar type, then it is ill-formed; so while c++98 does allow CLASS::operator->() returning pointer to scalar type, c++98 prohibits any '->'-expression involving this overloaded operator function.

Not only is this behaviour inconsistent, but also when comparing the corresponding chapters of c++pl2 and stdc++98 it looks like an oversight and unintended result. Mapping between stdc++98 and c++pl2:

c++pl2.r.5.2 -> 5.2 [expr.post]
c++pl2.r.5.2.4 -> 5.2.4 [expr.pseudo] + 5.2.5 [expr.ref]
c++pl2.r.13.4 -> 13.3.1.2 [over.match.oper]
c++pl2.r.13.4.6 -> 13.5.6 [over.ref]

For the single line of c++pl2.r.5.2 covering "->dtor", 5.2 [expr.post] has two lines. Analogously c++pl2.r.5.2.4 has been doubled to 5.2.4 [expr.pseudo] and 5.2.5 [expr.ref]. From 13.5.6 [over.ref], the sentence forbiding CLASS::operator->() returning pointer to scalar type has been removed. Only the single line of c++pl2.r.13.4.6 (<-> c++pl2.r.5.2's single line) has not gotten its 2nd line when converted into 13.5.6 [over.ref].

Additionally GCC32 does is right (but against 13.5.6 [over.ref]).

AFAICS this would not break old code except compilers like VC7x and Comeau4301.

It does not add new functionality, cause any expression class_type->scalar_type_dtor() even today can be substituted through (*class_type).scalar_type_dtor().

Without this fix, template functions like some_allocator<T>::destroy(p) must use "(*p).~T()" or "(*p).T::~T()" when calling the destructor, otherwise the simpler versions "p->~T()" or "p->T::~T()" could be used.

Sample code, compiled with GCC32, VC7[1] and Comeau4301:

```
struct A {};//any class

template <class T>
struct PTR
    {
    T& operator*  () const;
    T* operator-> () const;
    };

template <class T>
void f ()
    {
        {
        T*  p               ;
        p = new T           ;
        (*p).T::~T()        ;//OK
        p = new T           ;
        (*p).~T()           ;//OK
        p = new T           ;
        p->T::~T()          ;//OK
        p = new T           ;
        p->~T()             ;//OK
        }

        {
        PTR<T> p = PTR<T>() ;
        (*p).T::~T()        ;//OK
        (*p).~T()           ;//OK
        p.operator->()      ;//OK !!!
        p->T::~T()          ;//GCC32: OK; VC7x,Com4301: OK for A; ERROR w/ int
        p->~T()             ;//GCC32: OK; VC7x,Com4301: OK for A; ERROR w/ int
        }
    }

void test ()
    {
    f <A>  ();
    f <int>();
    }
```

Proposed resolution (April, 2005):

Change 16.5.6 [over.ref] paragraph 1 as indicated:

`operator->` shall be a non-static member function taking no parameters. It implements **the** class member access ~~using~~ **syntax that uses** `->`

postfix-expression `->` `template`***opt*** *id-expression*
postfix-expression `->` ***pseudo-destructor-name***

An expression `x->m` is interpreted as `(x.operator->())->m` for a class object `x` of type `T` if `T::operator->()` exists and if the operator is selected as the best match function by the overload resolution mechanism (16.3 [over.match]).

---

425. Set of candidates for overloaded built-in operator with float operand

Section: 16.6 [over.built] **Status:** CD1 **Submitter:** Daniel Frey **Date:** 30 June 2003

[Voted into WP at March 2004 meeting.]

During a discussion over at the boost mailing list (www.boost.org), we came across the following "puzzle":

```
struct A {
  template< typename T > operator T() const;
} a;

template<> A::operator float() const
{
  return 1.0f;
}

int main()
{
  float f = 1.0f * a;
}
```

The code is compiled without errors or warnings from EDG-based compilers (Comeau, Intel), but rejected from others (GCC, MSVC [7.1]). The question: Who is correct? Where should I file the bug report?

To explain the problem: The EDG seems to see 1.0f*a as a call to the unambiguous operator*(float,float) and thus casts 'a' to 'float'. The other compilers have several operators (float*float, float*double, float*int, ...) available and thus can't decide which cast is appropriate. I think the latter is the correct behaviour, but I'd like to hear some comments from the language lawyers about the standard's point of view on this problem.

Andreas Hommel: Our compiler also rejects this code:

```
Error   : function call 'operator*(float, {lval} A)' is ambiguous
'operator*(float, unsigned long long)'
'operator*(float, int)'
'operator*(float, unsigned int)'
'operator*(float, long)'
'operator*(float, unsigned long)'
'operator*(float, float)'
'operator*(float, double)'
'operator*(float, long double)'
'operator*(float, long long)'
Test.cp line 12       float f = 1.0f * a;
```

Is this example really legal? It was my understanding that all candidates from 16.6 [over.built] participate in overload resolution.

Daveed Vandevoorde: I believe the EDG-based compiler is right. Note that the built-in operator* requires "usual arithmetic conversions" (see 8.6 [expr.mul] paragraph 2 and 8 [expr] paragaph 9). This means that there is no candidate taking (float, double) arguments: Only (float, float) or (double, double).

Since your first argument is of type float, the (float, float) case is preferred over the (double, double) case (the latter would require a floating-point promotion).

Stave Adamczyk: Daveed's statement is wrong; as Andreas says, the prototypes in 16.6 [over.built] paragraph 12 have pairs of types, not the same type twice. However, the list of possibilities considered in Andreas' message is wrong also: 16.6 [over.built] paragraph 12 calls for pairs of **promoted** arithmetic types, and float is not a promoted type (it promotes to double -- see 7.7 [conv.fpprom]).

Nevertheless, the example is ambiguous. Let's look at the overload resolution costs. The right operand is always going to have a user-defined-conversion cost (the template conversion function will convert directly to the const version of the second parameter of the prototype). The left operand is always going to have a promotion (float --> double) or a standard conversion (anything else). So the cases with promotions are better than the others. However, there are several of those cases, with second parameters of type int, unsigned int, long, unsigned long, double, and long double, and all of those are equally good. Therefore the example is ambiguous.

Here's a reduced version that should be equivalent:

```
struct A {
  template <typename T> operator T() const;
} a;
void f(double, int);
void f(double, unsigned int);
void f(double, long);
void f(double, unsigned long);
void f(double, double);
void f(double, long double);
int main() {
```

```
  f(1.0f, a);  // Ambiguous
}
```

Personally, I think this is evidence that 16.6 [over.built] doesn't really do quite what it should. But the standard is clear, if possibly flawed.

Andreas Hommel: You are right, "float" is not a promoted arithmetic type, this is a bug in our compiler.

However, the usual arithmetic conversions (8 [expr] paragraph 9) do not promote the floating point types, so

```
float operator+(float, float);
```

is a legal built-in operator function, so I wonder if it shouldn't be included in the candidate list.

Steve Adamczyk: Hmm, the definition of the term in 16.6 [over.built] paragraph 2 is highly ambiguous:

> Similarly, the term promoted arithmetic type refers to promoted integral types plus floating types.

I can't tell if that's "promoted integral types plus (all) floating types" or "promoted integral types plus (promoted) floating types". I thought the latter was intended, but indeed the usual arithmetic conversions could give you "float + float", so it makes sense that float would be one of the possibilities. We should discuss this to make sure everyone has the same interpretation.

Proposed Resolution (October 2003):

Change the second sentence of 13.6 paragraph 2 as follows:

> Similarly, the term *promoted arithmetic type* refers to ~~promoted integral types plus floating types~~ **floating types plus promoted integral types**.

---

204. Exported class templates

Section: 17 [temp] **Status:** CD1 **Submitter:** Robert Klarer **Date:** 11 Feb 2000

[Voted into WP at April 2003 meeting.]

17 [temp] paragraph 7 allows class templates to be declared exported, including member classes and member class templates (implicitly by virtue of exporting the containing template class). However, paragraph 8 does not exclude exported class templates from the statement that

> An exported template need only be declared (and not necessarily defined) in a translation unit in which it is instantiated.

This is an incorrect implication; however, it is also not dispelled in 17.8.1 [temp.inst] paragraph 6:

> If an implicit instantiation of a class template specialization is required and the template is declared but not defined, the program is ill-formed.

This wording says nothing about the translation unit in which the definition must be provided. Contrast this with 17.8.2 [temp.explicit] paragraph 3:

> A definition of a class template or a class member template shall be in scope at the point of the explicit instantiation of the class template or class member template.

Suggested resolution:

- Change 17 [temp] paragraph 8 to say that "An exported **non-class** template need only be declared..."
- Change 17.8.1 [temp.inst] paragraph 6 to use wording similar to that of 17.8.2 [temp.explicit] paragraph 3 regarding the requirement for a definition of the class template to be in scope.

(See also issue 212.)

Notes from 04/00 meeting:

John Spicer opined that even though 17 [temp] paragraph 7 speaks of "declaring a class template exported," that does not mean that the class template is "an exported template" in the sense of paragraph 8. He suggested clarifying paragraph 7 to that effect instead of the change to paragraph 8 suggested above, and questioned the need for a change to 17.8.1 [temp.inst].

Notes from the 4/02 meeting:

This is resolved by the proposed changes for issue 323.

---

323. Where must `export` appear?

Section: 17 [temp] **Status:** CD1 **Submitter:** Daveed Vandevoorde **Date:** 14 Nov 2001

[Voted into WP at April 2003 meeting.]

The standard doesn't seem to describe whether the keyword `export` should appear on exported template declarations that are not used or defined in that particular translation unit.

For example:

```
// File 1:
template<typename T> void f();  // export omitted

// File 2:
export template<typename T> void f() {}

int main() { f<int>(); }
```

Another example is:

```
// File 1:
struct S {
   template<typename T> void m();
};

// File 2:
struct S {
   template<typename T> void m();
};

export template<typename T> void S::m() {}

int main() {
   S s;
   S.m<int>();
}
```

I think both examples should be clarified to be invalid. If a template is exported in one translation unit, it should be declared export in all translation units in which it appears.

With the current wording, it seems that even the following is valid:

```
// File 1:
export template<typename T> void f();  // export effectively ignored

// File 2:
template<typename T> void f() {}  // Inclusion model
void g() { f<int>(); }

// File 3:
void g();
template<typename T> void f() {}  // Inclusion model

int main() {
   g();
   f<int>();
}
```

In fact, I think the declaration in "File 1" could be a definition and this would still satisfy the the requirements of the standard, which definitely seems wrong.

Proposed Resolution (revised October 2002):

Replace 17 [temp] paragraphs 6, 7, and 8 by the following text:

> A *template-declaration* may be preceded by the `export` keyword. Such a template is said to be *exported*. Declaring exported a class template is equivalent to declaring exported all of its non-inline member functions, static data members, member classes, member class templates, and non-inline member function templates.
>
> If a template is exported in one translation unit, it shall be exported in all translation units in which it appears; no diagnostic is required. A declaration of an exported template shall appear with the `export` keyword before any point of instantiation (17.7.4.1 [temp.point]) of that template in that translation unit. In addition, the first declaration of an exported template containing the `export` keyword must not follow the definition of that template. The `export` keyword shall not be used in a friend declaration.
>
> Templates defined in an unnamed namespace, inline functions, and inline function templates shall not be exported. An exported non-class template shall be defined only once in a program; no diagnostic is required. An exported non-class template need only be declared (and not necessarily defined) in a translation unit in which it is instantiated.
>
> A non-exported non-class template must be defined in every translation unit in which it is implicitly instantiated (17.8.1 [temp.inst]), unless the corresponding specialization is explicitly instantiated (17.8.2 [temp.explicit]) in some translation unit; no diagnostic is required.

Note: This change also resolves issues 204 and 335.

---

335. Allowing `export` on template members of nontemplate classes

Section: 17 [temp] **Status:** CD1 **Submitter:** John Spicer **Date:** 30 Jan 2002

[Voted into WP at April 2003 meeting.]

The syntax for "export" permits it only on template declarations. Clause 17 [temp] paragraph 6 further restricts "export" to appear only on namespace scope declarations. This means that you can't export a member template of a non-template class, as in:

```
class A {
  template <class T> void f(T);
};
```

You can, of course, put export on the definition:

```
export template <class T> void A<T>::f(T){}
```

but in order for the template to be used from other translation units (the whole point of export) the declaration in the other translation unit must also be declared export.

There is also the issue of whether or not we should permit this usage:

```
export struct A {
  template <class T> void f(T);
};
```

My initial reaction is to retain this prohibition as all current uses of "export" are preceding the "template" keyword.

If we eliminate the requirement that "export" precede "template" there is a similar issue regarding this case, which is currently prohibited:

```
template <class T> struct B {
  export void f();
};
```

My preference is still to permit only "export template".

Notes from the 4/02 meeting:

This is resolved by the proposed changes for issue 323.

---

534. *template-name*s and *operator-function-id*s

Section: 17 [temp] **Status:** CD1 **Submitter:** Jens Maurer **Date:** 5 October 2005

[Voted into WP at the October, 2006 meeting.]

Taken literally, 17 [temp] paragraph 2 does not permit operator functions to be templates:

> In a function template declaration, the *declarator-id* shall be a *template-name* (i.e., not a *template-id*).

and, in 17.2 [temp.names] paragraph 1, a *template-name* is defined to be simply an *identifier*.

Issue 301 considered and rejected the idea of changing the definition of *template-name* to include *operator-function-id*s and *conversion-function-id*s. Either that decision should be reconsidered or the various references in the text to *template-name* should be examined to determine if they should also mention the non-*identifier* possibilities for function template names.

Proposed resolution (April, 2006):

This issue is resolved by the resolution of issue 301.

---

184. Default arguments in template *template-parameter*s

Section: 17.1 [temp.param] **Status:** CD1 **Submitter:** John Spicer **Date:** 11 Nov 1999

[Voted into WP at April 2003 meeting.]

John Spicer: Where can default values for the template parameters of template template parameters be specified and where so they apply?

For normal template parameters, defaults can be specified only in class template declarations and definitions, and they accumulate across multiple declarations in the same way that function default arguments do.

I think that defaults for parameters of template template parameters should be handled differently, though. I see no reason why such a default should extend beyond the template declaration with which it is associated. In other words, such defaults are a property of a specific template declaration and are not part of the interface of the template.

```
template <class T = float> struct B {};

template <template <class _T = float> class T> struct A {
    inline void f();
    inline void g();
};

template <template <class _T> class T> void A<T>::f() {
    T<> t;  // Okay? (proposed answer - no)
}
```

```
template <template <class _T = char> class T> // Okay? (proposed answer - yes)
void A<T>::g() {
    T<> t;  // T<char> or T<float>?  (proposed answer - T<char>)
}

int main() {
    A<B> ab;
    ab.f();
}
```

I don't think this is clear in the standard.

Gabriel Dos Reis: On the other hand I fail to see the reasons why we should introduce yet another special rule to handle that situation differently. I think we should try to keep rules as uniform as possible. For default values, it has been the case that one should look for any declaration specifying default values. Breaking that rules doesn't buy us anything, at least as far as I can see. My feeling is that [allowing different defaults in different declarations] is very confusing.

Mike Miller: I'm with John on this one. Although we don't have the concept of "prototype scope" for template parameter lists, the analogy with function parameters would suggest that the two declarations of `T` (in the template class definition and the template member function definition) are separate declarations and completely unrelated. While it's true that you accumulate default arguments on top-level declarations in the same scope, it seems to me a far leap to say that we ought also to accumulate default arguments in nested declarations. I would expect those to be treated as being in different scopes and thus **not** to share default argument information.

When you look up the name `T` in the definition of `A<T>::f()`, the declaration you find has no default argument for the parameter of `T`, so `T<>` should not be allowed.

Proposed Resolution (revised October 2002):

In 17.1 [temp.param], add the following as a new paragraph at the end of this section:

> A *template-parameter* of a template *template-parameter* is permitted to have a default *template-argument*. When such default arguments are specified, they apply to the template *template-parameter* in the scope of the template *template-parameter*. [*Example:*
>
> ```
> template <class T = float> struct B {};
>
> template <template <class TT = float> class T> struct A {
>     inline void f();
>     inline void g();
> };
>
> template <template <class TT> class T> void A<T>::f() {
>     T<> t;  // error - TT has no default template argument
> }
>
> template <template <class TT = char> class T>void A<T>::g() {
>     T<> t;  // OK - T<char>
> }
> ```
>
> -- *end example*]

---

215. Template parameters are not allowed in *nested-name-specifier*s

Section: 17.1 [temp.param] **Status:** CD1 **Submitter:** Martin von Loewis **Date:** 13 Mar 2000

[Voted into WP at April, 2007 meeting.]

According to 17.1 [temp.param] paragraph 3, the following fragment is ill-formed:

```
template <class T>
class X{
  friend void T::foo();
};
```

In the friend declaration, the `T::` part is a *nested-name-specifier* (11 [dcl.decl] paragraph 4), and `T` must be a *class-name* or a *namespace-name* (_N4567_.5.1.1 [expr.prim.general] paragraph 7). However, according to 17.1 [temp.param] paragraph 3, it is only a *type-name*. The fragment should be well-formed, and instantiations of the template allowed as long as the actual template argument is a class which provides a function member `foo`. As a result of this defect, any usage of template parameters in nested names is ill-formed, e.g., in the example of 17.7 [temp.res] paragraph 2.

Notes from 04/00 meeting:

The discussion at the meeting revealed a self-contradiction in the current IS in the description of *nested-name-specifier*s. According to the grammar in _N4567_.5.1.1 [expr.prim.general] paragraph 7, the components of a *nested-name-specifier* must be either *class-name*s or *namespace-name*s, i.e., the constraint is syntactic rather than semantic. On the other hand, 6.4.3 [basic.lookup.qual] paragraph 1 describes a semantic constraint: only object, function, and enumerator names are ignored in the lookup for the component, and the program is ill-formed if the lookup finds anything other than a *class-name* or *namespace-name*. It was generally agreed that the syntactic constraint should be eliminated, i.e., that the grammar ought to be changed not to use *class-or-namespace-name*.

A related point is the explicit prohibition of use of template parameters in *elaborated-type-specifier*s in 10.1.7.3 [dcl.type.elab] paragraph 2. This rule was the result of an explicit Committee decision and should not be unintentionally voided by the resolution of this issue.

Proposed resolution (04/01):

Change _N4567_.5.1.1 [expr.prim.general] paragraph 7 and A.4 [gram.expr] from

nested-name-specifier:
class-or-namespace-name `::` *nested-name-specifier*$_{opt}$
class-or-namespace-name `::` `template` *nested-name-specifier*

class-or-namespace-name:
class-name
namespace-name

to

nested-name-specifier:
type-or-namespace-name `::` *nested-name-specifier*$_{opt}$
type-or-namespace-name `::` `template` *nested-name-specifier*

type-or-namespace-name:
type-name
namespace-name

This resolution depends on the resolutions for issues 245 (to change the name lookup rules in *elaborated-type-specifier*s to include all *type-name*s) and 283 (to categorize template *type-parameter*s as *type-name*s).

Notes from 10/01 meeting:

There was some sentiment for going with simply *identifier* in front of the "`::`", and stronger sentiment for going with something with a more descriptive name if possible. See also issue 180.

Notes from April 2003 meeting:

This was partly resolved by the changes for issue 125. However, we also need to add a semantic check in 6.4.3 [basic.lookup.qual] to allow `T::foo` and we need to reword the first sentence of 6.4.3 [basic.lookup.qual].

Proposed resolution (October, 2004):

Change 6.4.3 [basic.lookup.qual] paragraph 1 as follows:

> The name of a class or namespace member can be referred to after the `::` scope resolution operator (_N4567_.5.1.1 [expr.prim.general]) applied to a *nested-name-specifier* that nominates its class or namespace. During the lookup for a name preceding the `::` scope resolution operator, object, function, and enumerator names are ignored. If the name found ~~*is not a class-name* (clause 12 [class]) or *namespace-name* (10.3.1 [namespace.def])~~ **does not designate a class or namespace**, the program is ill-formed. [...]

Notes from the April, 2005 meeting:

The 10/2004 resolution does not take into account the fact that template type parameters do not designate class types in the context of the template definition. Further drafting is required.

Proposed resolution (April, 2006):

Change 6.4.3 [basic.lookup.qual] paragraph 1 as follows:

> The name of a class or namespace member can be referred to after the `::` scope resolution operator (_N4567_.5.1.1 [expr.prim.general]) applied to a *nested-name-specifier* that nominates its class or namespace. During the lookup for a name preceding the `::` scope resolution operator, object, function, and enumerator names are ignored. If the name found ~~*is not a class-name* (clause 12 [class]) or *namespace-name* (10.3.1 [namespace.def])~~ **does not designate a namespace or a class or dependent type**, the program is ill-formed. [...]

---

226. Default template arguments for function templates

Section: 17.1 [temp.param] **Status:** CD1 **Submitter:** Bjarne Stroustrup **Date:** 19 Apr 2000

[Voted into WP at April 2003 meeting.]

The prohibition of default template arguments for function templates is a misbegotten remnant of the time where freestanding functions were treated as second class citizens and required all template arguments to be deduced from the function arguments rather than specified.

The restriction seriously cramps programming style by unnecessarily making freestanding functions different from member functions, thus making it harder to write STL-style code.

Suggested resolution:

Replace

A default *template-argument* shall not be specified in a function template declaration or a function template definition, nor in the *template-parameter-list* of the definition of a member of a class template.

by

A default *template-argument* shall not be specified in the *template-parameter-list* of the definition of a member of a class template.

The actual rules are as stated for arguments to class templates.

Notes from 10/00 meeting:

The core language working group was amenable to this change. Questions arose, however, over the interaction between default template arguments and template argument deduction: should it be allowed or forbidden to specify a default argument for a deduced parameter? If it is allowed, what is the meaning: should one or the other have priority, or is it an error if the default and deduced arguments are different?

Notes from the 10/01 meeting:

It was decided that default arguments should be allowed on friend declarations only when the declaration is a definition. It was also noted that it is not necessary to insist that if there is a default argument for a given parameter all following parameters have default arguments, because (unlike in the class case) arguments can be deduced if they are not specified.

Note that there is an interaction with issue 115.

Proposed resolution (revised October 2002):

1. In 17.1 [temp.param] paragraph 9, replace

   A default *template-argument* may be specified in a class template declaration or a class template definition. A default *template-argument* shall not be specified in a function template declaration or a function template definition, nor in the *template-parameter-list* of the definition of a member of a class template.

   with

   A default *template-argument* may be specified in a template declaration. A default *template-argument* shall not be specified in the *template-parameter-list*s of the definition of a member of a class template that appears outside of the member's class.

2. In 17.1 [temp.param] paragraph 9, replace

   A default *template-argument* shall not be specified in a friend template declaration.

   with

   A default *template-argument* shall not be specified in a friend class template declaration. If a friend function template declaration specifies a default *template-argument*, that declaration shall be a definition and shall be the only declaration of the function template in the translation unit.

3. In 17.1 [temp.param] paragraph 11, replace

   If a *template-parameter* has a default *template-argument*, all subsequent *template-parameters* shall have a default *template-argument* supplied.

   with

   If a *template-parameter* of a class template has a default *template-argument*, all subsequent *template-parameters* shall have a default *template-argument* supplied. [*Note:* This is not a requirement for function templates because template arguments might be deduced (17.9.2 [temp.deduct]).]

4. In 17.9 [temp.fct.spec] paragraph 1, replace

   Template arguments can either be explicitly specified when naming the function template specialization or be deduced (17.9.2 [temp.deduct]) from the context, e.g. from the function arguments in a call to the function template specialization.

   with

   Template arguments can be explicitly specified when naming the function template specialization, deduced from the context (17.9.2 [temp.deduct]), e.g., deduced from the function arguments in a call to the function template specialization), or obtained from default template arguments.

5. In 17.9.1 [temp.arg.explicit] paragraph 2, replace

   Trailing template arguments that can be deduced (17.9.2 [temp.deduct]) may be omitted from the list of explicit *template-argument*s.

   with

   Trailing template arguments that can be deduced (17.9.2 [temp.deduct]) or obtained from default *template-argument*s may be omitted from the list of explicit *template-argument*s.

6. In 17.9.2 [temp.deduct] paragraph 1, replace

   The values can be either explicitly specified or, in some cases, deduced from the use.

   with

   The values can be explicitly specified or, in some cases, be deduced from the use or obtained from default *template-argument*s.

7. In 17.9.2 [temp.deduct] paragraph 4, replace

   The resulting substituted and adjusted function type is used as the type of the function template for template argument deduction. When all template arguments have been deduced, all uses of template parameters in nondeduced contexts are replaced with the corresponding deduced argument values. If the substitution results in an invalid type, as described above, type deduction fails.

   with

   The resulting substituted and adjusted function type is used as the type of the function template for template argument deduction. If a template argument has not been deduced, its default template argument, if any, is used. [*Example:*

```
template <class T, class U = double>
void f(T t = 0, U u = 0);

void g()
{
    f(1, 'c');          // f<int,char>(1,'c')
    f(1)                // f<int,double>(1,0)
    f();                // error: T cannot be deduced
    f<int>();           // f<int,double>(0,0)
    f<int,char>();      // f<int,char>(0,0)
}
```

   ---*end example*]

   When all template arguments have been deduced or obtained from default template arguments, all uses of template parameters in nondeduced contexts are replaced with the corresponding deduced or default argument values. If the substitution results in an invalid type, as described above, type deduction fails.

---

401. When is access for template parameter default arguments checked?

Section: 17.1 [temp.param] **Status:** CD1 **Submitter:** Steve Adamczyk **Date:** 27 Jan 2003

[Voted into WP at October 2005 meeting.]

Is the following well-formed?

```
class policy {};
class policy_interface {};
template <class POLICY_INTERFACE>
class aph {
protected:
  typedef POLICY_INTERFACE PI;
};
template <class POLICY, class BASE, class PI = typename BASE::PI>
class ConcretePolicyHolder : public BASE, protected POLICY
{};
ConcretePolicyHolder < policy , aph < policy_interface > > foo;
void xx() { }
```

The issue is whether the access to the default argument type BASE::PI is checked before or after it is known that BASE is a base class of the template. To some extent, one needs to develop the list of template arguments (and therefore evaluate the default argument) before one can instantiate the template, and one does not know what base classes the template has until it has been instantiated.

Notes from April 2003 meeting:

Shortened example:

```
class B {
protected:
  typedef int A;
};
template<class T, class U = typename T::A>
class X : public T
{ };
```

The convincing argument here is that if we had only the declaration of the template (including the default argument), we would expect it to be usable in exactly the same way as the version with the definition. However, the special access needed is visible only when the definition is available. So the above should be an error, and information from the definition cannot affect the access of the default arguments.

Proposed Resolution (April 2003):

Add a new paragraph 16 to 17.1 [temp.param] after paragraph 15:

Since a default *template-argument* is encountered before any *base-clause* there is no special access to members used in a default *template-argument*. *[Example:*

```
class B {};
template <class T> class C {
protected:
   typedef T TT;
};

template <class U, class V = typename U::TT>
class D : public U {};

D <C<B> > d;  // access error, C::TT is protected
```

--- end example]

Notes from October 2003 meeting:

We decided that template parameter default arguments should have their access checked in the context where they appear without special access for the entity declared (i.e., they are different than normal function default arguments). One reason: we don't know the instance of the template when we need the value. Second reason: compilers want to parse and throw away the form of the template parameter default argument, not save it and check it for each instantiation.

Class templates should be treated the same as function templates in this regard. The base class information is in the same category as friend declarations inside the class itself -- not available. If the body were used one would need to instantiate it in order to know whether one can name it.

Proposed resolution (October, 2004):

Add the following as a new paragraph following the last paragraph of 14 [class.access] (but before the new paragraph inserted by the resolution of issue 372, if adopted):

The names in a default *template-argument* (17.1 [temp.param]) have their access checked in the context in which they appear rather than at any points of use of the default *template-argument*. [*Example:*

```
class B {};
template <class T> class C {
protected:
   typedef T TT;
};

template <class U, class V = typename U::TT>
class D : public U {};

D <C<B> >* d;  // access error, C::TT is protected
```

—*end example*]

---

228. Use of `template` keyword with non-member templates

Section: 17.2 [temp.names] **Status:** CD1 **Submitter:** Daveed Vandevoorde **Date:** 4 May 2000

[Voted into WP at April 2003 meeting.]

Consider the following example:

```
template<class T>
struct X {
   virtual void f();
};

template<class T>
struct Y {
   void g(X<T> *p) {
      p->template X<T>::f();
   }
};
```

This is an error because `X` is not a member template; 17.2 [temp.names] paragraph 5 says:

> If a name prefixed by the keyword `template` is not the name of a member template, the program is ill-formed.

In a way this makes perfect sense: `X` is found to be a template using ordinary lookup even though `p` has a dependent type. However, I think this makes the use of the `template` prefix even harder to teach.

Was this intentionally outlawed?

Proposed Resolution (4/02):

Elide the first use of the word "member" in 17.2 [temp.names] paragraph 5 so that its first sentence reads:

> If a name prefixed by the keyword `template` is not the name of a ~~member~~ template, the program is ill-formed.

---

301. Syntax for *template-name*

Section: 17.2 [temp.names] **Status:** CD1 **Submitter:** Mark Mitchell **Date:** 24 Jul 2001

[Voted into WP at the October, 2006 meeting.]

The grammar for a *template-name* is:

> *template-name*:
> *identifier*

That's not right; consider:

```
template <class T> T operator+(const T&, const T&);
template <> S operator+<S>(const S&, const S&);
```

This is ill-formed according to the standard, since `operator+` is not a template-name.

Suggested resolution:

I think the right rule is

> *template-name:*
> *identifier*
> *operator-function-id*
> *conversion-function-id*

John Spicer adds that there's some question about whether conversion functions should be included, as they cannot have template argument lists.

Notes from 4/02 meeting:

If the change is made as a syntax change, we'll need a semantic restriction to avoid `operator+<int>` as a class. Clark Nelson will work on a compromise proposal -- not the minimal change to the syntax proposed, not the maximal change either.

Clark Nelson (April 2003):

The proposed solution (adding *operator-function-id* as an alternative for *template-name*) would have a large impact on the language described by the grammar. Specifically, for example, `operator+<int>` would become a syntactically valid *class-name*.

On the other hand, a change with (I believe) exactly the desired effect on the language accepted, would be to modify *operator-function-id* itself:

> *operator-function-id*:
> `operator` *operator*
> `operator` ***operator*** `<` ***template-argument-list*$_{opt}$** `>`

(Steve Adamczyk: this change was already made by issue 38 and is in TC1.)

Then there is the first sentence of 17.2 [temp.names] paragraph 3:

> After name lookup (6.4 [basic.lookup]) finds that a name is a *template-name*, if this name is followed by a `<`, the `<` is always taken as the beginning of a *template-argument-list* and never as a name followed by the less-than operator.

This description seems to be adequate for names of class templates. As far as I can tell, the only ambiguity it resolves is from something that starts with `new X <`, in the scope of a class template `X`. But as far as I can tell is already inadequate for names of function templates, and is even worse for operator function templates.

Probably `<` should always be interpreted as introducing a *template-argument-list* if any member of the overload set is a function template. After all, function pointers are very rarely compared for ordering, and it's not clear what other rule might be workable.

I'm inclined to propose the simplest rule possible for *operator-function-ids*: if one is followed by `<`, then what follows is interpreted as a *template-argument-list*, unconditionally. Of course, if no template for that operator has been declared, then there's an error.

Also, note that if the operator in question is `<` or `<<`, it is possible to run into a problem similar to the famous `>>` nested template argument list closing delimiter problem. However, since in this case (at least) one of the `<` characters has a radically different interpretation than the other, and for other reasons as well, this is unlikely to be nearly as much of a practical problem as the `>>` problem.

Notes from April 2003 meeting:

We felt that the operator functions should not be special-cased. They should be treated like any other name.

September 2003:

Clark Nelson has provided the changes in N1490=03-0073.

Notes from October 2003 meeting:

We reviewed Clark Nelson's N1490. Clark will revise it and introduce a new syntax term for an identifier or the name of an operator function.

Notes from the April, 2005 meeting:

The CWG suggested a new approach to resolving this issue: the existing term *template-id* will be renamed to *class-template-id*, the term *template-id* will be defined to include operator functions with template arguments, and any current uses of *template-id* (such as in the definition of *elaborated-type-specifier*) where an operator function is not appropriate will be changed to refer to *class-template-id*.

Proposed resolution (April, 2006):

As specified in document J16/05-0156 = WG21 N1896, except that:

1. In change 9 (6.4.5 [basic.lookup.classref]), omit the change from "entire *postfix-expression*" to "*nested-name-specifier*."
2. In change a (6.4.3.1 [class.qual] paragraph 1, third bullet), omit the change from "entire *postfix-expression*" to "*qualified-id*."
3. In change b (6.4.3.2 [namespace.qual] paragraph 1), omit the change from "entire *postfix-expression*" to "*qualified-id*."

(Some of these omitted changes are addressed by issue 562.)

(This resolution also resolves issue 534.)

---

468. Allow `::template` outside of templates

Section: 17.2 [temp.names] **Status:** CD1 **Submitter:** John Spicer **Date:** 9 Apr 2004

[Voted into WP at the October, 2006 meeting.]

For the same reasons that issue 382 proposes for relaxation of the requirements on `typename`, it would make sense to allow the `::template` disambiguator outside of templates.

See also issues 11, 30, 96, and 109.

Proposed resolution (October, 2005):

Change 17.2 [temp.names] paragraph 5 as indicated:

> If a name prefixed by the keyword `template` is not the name of a template, the program is ill-formed. [*Note:* the keyword `template` may not be applied to non-template members of class templates. —*end note*] ~~Furthermore, names of member templates shall not be prefixed by the keyword `template` if the *postfix-expression* or *qualified-id* does not appear in the scope of a template.~~ [*Note:* just as is the case with the `typename` prefix, the `template` prefix is allowed in cases where it is not strictly necessary; i.e., when **the *nested-name-specifier or*** the expression on the left of the `->` or `.` ~~, or the *nested-name-specifier*~~ is not dependent on a *template-parameter*, **or the use does not appear in the scope of a template**. —*end note*]

---

246. Jumps in *function-try-block* handlers

Section: 17.3 [temp.arg] **Status:** CD1 **Submitter:** Mike Miller **Date:** 15 Sep 2000

[Moved to DR at 4/01 meeting.]

Is it permitted to jump from a handler of a *function-try-block* into the body of the function?

18 [except] paragraph 2 would appear to disallow such a jump:

> A `goto`, `break`, `return`, or `continue` statement can be used to transfer control out of a try block or handler, but not into one.

However, 18.3 [except.handle] paragraph 14 mentions only constructors and destructors for the prohibition:

> If the handlers of a *function-try-block* contain a jump into the body of a constructor or destructor, the program is ill-formed.

Is this paragraph simply reemphasizing the more general restriction, or does it assume that such a jump would be permitted for functions other than constructors or destructors? If the former interpretation is correct, it is confusing and should be either eliminated or turned into a note. If the latter interpretation is accurate, 18 [except] paragraph 2 must be revised.

(See also issue 98.)

Proposed resolution (04/01):

Delete 18.3 [except.handle] paragraph 14.

---

372. Is access granted by base class specifiers available in following base class specifiers?

Section: 17.3 [temp.arg] **Status:** CD1 **Submitter:** Clark Nelson **Date:** 13 August 2002

[Voted into WP at the October, 2006 meeting.]

I'm not really sure what the standard says about this. Personally, I'd like for it to be ill-formed, but I can't find any words that I can interpret to say so.

```
template<class T>
class X
{
protected:
  typedef T Type;
};
template<class T>
class Y
{
};
template<class T,
         template<class> class T1,
         template<class> class T2>
class Z:
  public T2<typename T1<T>::Type>,
  public T1<T>
{
};
Z<int, X, Y> z;
```

John Spicer: I don't think the standard really addresses this case. There is wording about access checking of things used as template arguments, but that doesn't address accessing members of the template argument type (or template) from within the template.

This example is similar, but does not use template template arguments.

```
class X {
private:
  struct Type {};
};
template <class T> struct A {
  typename T::Type t;
};
A<X> ax;
```

This gets an error from most compilers, though the standard is probably mute on this as well. An error makes sense -- if there is no error, there is a hole in the access checking. (The special rule about no access checks on template parameters is not a hole, because the access is checked on the type passed in as an argument. But when you look up something in the scope of a template parameter type, you need to check the access to the member found.)

The logic in the template template parameter case should be similar: anytime you look up something in a template-dependent class, the member's access must be checked, because it could be different for different template instances.

Proposed Resolution (October 2002):

Change the last sentence of 17.3 [temp.arg] paragraph 3 from:

> For a *template-argument* of class type, the template definition has no special access rights to the inaccessible members of the template argument type.

to:

> For a *template-argument* that is a class type or a class template, the template definition has no special access rights to the members of the *template-argument*. [*Example*:
>
> ```
> template <template <class TT> class T> class A {
>   typename T<int>::S s;
> };
>
> template <class U> class B {
>   private:
>   struct S { /* ... */ };
> };
>
> A<B> b;   // ill-formed, A has no access to B::S
> ```
>
> -- *end example*]

Daniel Frey posts on comp.std.c++ in July 2003: I just read DR 372 and I think that the problem presented is not really discussed/solved properly. Consider this example:

```
class A {
protected:
   typedef int N;
};

template< typename T >
class B
{};

template< typename U >
class C : public U, public B< typename U::N >
{};

C< A > x;
```

The question is: If C is derived from A as above, is it allowed to access A::N before the classes opening '{'?

The main problem is that you need to access U's protected parts in C's base-clause. This pattern is common when using policies, Andrei's Loki library was bitten by it as he tried to make some parts of the policies 'protected' but some compilers rejected the code. They were right to reject it, I think it's 14.3 [class.friend]/2 that applies here and prevents the code above to be legal, although it addresses a different and reasonable example. To me, it seems wrong to reject the code as it is perfectly reasonable to write such stuff. The questions are:

- Do you agree it's reasonable?
- Is it a DR or is it a request for an extension?
- Is DR 372 the right place to address it or shall it be a new DR?

Steve Adamczyk: In other words, the point of the issue is over what range access derived from base class specifiers is granted, and whether any part of that range is the base specifier list itself, either the parts afterwards or the whole base specifier list. (Clark Nelson confirms this is what he was asking with the original question.) Personally, I find it somewhat disturbing that access might arrive incrementally; I'd prefer that the access happen all at once, at the opening brace of the class.

Notes from October 2003 meeting:

We decided it makes sense to delay the access checking for the base class specifiers until the opening brace of the class is seen. In other words, the base specifiers will be checked using the full access available for the class, and the order of the base classes is not significant in that determination. The implementors present all said they already had code to handle accumulation of delayed access checks, because it is already needed in other contexts.

Proposed resolution (October, 2004):

1. Change the last sentence of 17.3 [temp.arg] paragraph 3 as indicated:

   For a *template-argument* ~~of~~ **that is a** class type **or a class template**, the template definition has no special access rights to the ~~inaccessible~~ members of the ~~template argument type~~ ***template-argument*****. [*****Example:***

   ```
   template <template <class TT> class T> class A {
      typename T<int>::S s;
   };

   template <class U> class B {
      private:
      struct S { /* ... */ };
   };

   A<B> b;   // ill-formed, A has no access to B::S
   ```

   —end example**]**

2. Insert the following as a new paragraph after the final paragraph of 14 [class.access]:

   The access of names in a class *base-specifier-list* are checked at the end of the list after all base classes are known. [*Example:*

   ```
   class A {
     protected:
       typedef int N;
   };

   template<typename T> class B {};

   template<typename U> class C : public B<typename U::N>, public U {};

   C<A> x;  // OK: A is a base class so A::N in B<A::N> is accessible
   ```

—end example]

Notes from the April, 2005 meeting:

The 10/2004 resolution is not sufficient to implement the CWG's intent to allow these examples: clause 14 [class.access] paragraph 1 grants protected access only to "members and friends" of derived classes, not to their *base-specifier*s. The resolution needs to be extended to say either that access in *base-specifiers* is determined as if they were members of the class being defined or that access is granted to the class as an entity, including its *base-specifier*s. See also issue 500, which touches on the same issue and should be resolved in the same way.

Proposed resolution (October, 2005):

1. Change the second bullet of 14 [class.access] paragraph 1 as indicated:

   - `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~~ **by classes derived from that class, and by their friends** (see 14.4 [class.protected]).

2. Change 14 [class.access] paragraph 2 as indicated:

   A member of a class can also access all the names ~~declared in the class of which it is a member~~ **to which the class has access**.

3. Change 14 [class.access] paragraph 6 as indicated:

   All access controls in clause 14 [class.access] affect the ability to access a class member name from a particular scope. ~~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.~~ **For purposes of access control, the**

***base-specifier*s of a class and the definitions of class members that appear outside of the class definition are considered to be within the scope of that class.** In particular...

4. Change the example and commentary in 14 [class.access] paragraphs 6-7 as indicated:

[*Example:*

```
class A {
  typedef int I; // private member
  I f();
  friend I g(I);
  static I x;
protected:
   struct B { };
};

A::I A::f () { return 0; }
A::I g(A::I p = A::x);
A::I g(A::I p) { return 0; }
A::I A::x = 0;

struct D: A::B, A { };
```

Here, all the uses of `A::I` are well-formed because `A::f` and `A::x` are members of class `A` and `g` is a friend of class `A`. This implies, for example, that access checking on the first use of `A::I` must be deferred until it is determined that this use of `A::I` is as the return type of a member of class `A`. **Similarly, the use of `A::B` as a *base-specifier* is well-formed because `D` is derived from `A`, so access checking of *base-specifier*s must be deferred until the entire *base-specifier-list* has been seen.** —*end example*]

5. In 14.3 [class.friend] paragraph 2, replace the following text:

Declaring a class to be a friend implies that the names of private and protected members from the class granting friendship can be accessed in declarations of members of the befriended class. [*Note:* this means that access to private and protected names is also granted to member functions of the friend class (as if the functions were each friends) and to the static data member definitions of the friend class. This also means that private and protected type names from the class granting friendship can be used in the *base-clause* of a nested class of the friend class. However, the declarations of members of classes nested within the friend class cannot access the names of private and protected members from the class granting friendship. Also, because the *base-clause* of the friend class is not part of its member declarations, the *base-clause* of the friend class cannot access the names of the private and protected members from the class granting friendship. For example,

```
class A {
  class B { };
  friend class X;
};
class X: A::B {     // ill-formed: A::B cannot be accessed
                    // in the base-clause for X
  A::B mx;          // OK: A::B used to declare member of X
  class Y: A::B {   // OK: A::B used to declare member of X
    A::B my;        // ill-formed: A::B cannot be accessed
                    // to declare members of nested class of X
  };
};
```

—*end note*]

with:

Declaring a class to be a friend implies that the names of private and protected members from the class granting friendship can be accessed in the *base-specifier*s and member declarations of the befriended class. [*Example:*

```
class A {
  class B { };
  friend class X;
};

struct X: A::B {   // OK: A::B accessible to friend
  A::B mx;         // OK: A::B accessible to member of friend
  class Y {
    A::B my;       // OK: A::B accessible to nested member of friend
  };
};
```

—*end example*]

6. Change the last sentence of 17.3 [temp.arg] paragraph 3 as indicated:

For a *template-argument* ~~of~~ **that is a** class type **or a class template**, the template definition has no special access rights to the ~~inaccessible~~ members of the ~~template argument type.~~ ***template-argument*. [*Example:***

```
template <template <class TT> class T> class A {
  typename T<int>::S s;
};

template <class U> class B {
private:
  struct S { /* ... */ };
};

A<B> b;     // ill-formed, A has no access to B::S
```

—end example]

7. Change 12.2.5 [class.nest] paragraph 4 as indicated:

> Like a member function, a friend function (14.3 [class.friend]) defined within a nested class is in the lexical scope of that class; it obeys the same rules for name binding as a static member function of that class (12.2.3 [class.static])**, but it** ~~and~~ has no special access rights to members of an enclosing class.

(Note: this resolution also resolves issues 494 and 500.)

---

62. Unnamed members of classes used as type parameters

Section: 17.3.1 [temp.arg.type] **Status:** CD1 **Submitter:** Steve Adamczyk **Date:** 13 Oct 1998

[Moved to DR at 4/01 meeting.]

Section 17.3.1 [temp.arg.type] paragraph 2 says

> A local type, a type with no linkage, an unnamed type or a type compounded from any of these types shall not be used as a *template-argument* for a template *type-parameter*.

It probably wasn't intended that classes with unnamed members should be included in this list, but they are arguably compounded from unnamed types.

Proposed resolution (04/01):

In 17.3.1 [temp.arg.type] paragraph 2, change

> A local type, a type with no linkage, an unnamed type or a type compounded from any of these types shall not be used as a *template-argument* for a template *type-parameter*.

to

> The following types shall not be used as a *template-argument* for a template *type-parameter*:
>
> - a type whose name has no linkage
> - an unnamed class or enumeration type that has no name for linkage purposes (10.1.3 [dcl.typedef])
> - a cv-qualified version of one of the types in this list
> - a type created by application of declarator operators to one of the types in this list
> - a function type that uses one of the types in this list

---

354. Null as nontype template argument

Section: 17.3.2 [temp.arg.nontype] **Status:** CD1 **Submitter:** John Spicer **Date:** 2 May 2002

[Voted into WP at October 2005 meeting.]

The standard does not permit a null value to be used as a nontype template argument for a nontype template parameter that is a pointer.

This code is accepted by EDG, Microsoft, Borland and Cfront, but rejected by g++ and Sun:

```
template <int *p> struct A {};
A<(int*)0> ai;
```

I'm not sure this was ever explicitly considered by the committee. Is there any reason to permit this kind of usage?

Jason Merrill: I suppose it might be useful for a program to be able to express a degenerate case using a null template argument. I think allowing it would be harmless.

Notes from October 2004 meeting:

CWG decided that it would be desirable to allow null pointers as nontype template arguments, even though they are not representable in some current ABIs. There was some discussion over whether to allow a bare `0` to be used with a pointer nontype template parameter. The following case was decisive:

```
template<int i> void foo();
template<int* i> void foo();
...
foo<0>();
```

The current wording of 17.3 [temp.arg] paragraph 7 disambiguates the function call in favor of the `int` version. If the null pointer conversion were allowed for pointer nontype template parameters, this case would become ambiguous, so it was decided to require a cast.

Proposed resolution (April, 2005):

1. In 17.3.2 [temp.arg.nontype] paragraph 1, insert the following after the third bullet:

   - a constant expression that evaluates to a null pointer value (7.11 [conv.ptr]); or
   - a constant expression that evaluates to a null member pointer value (7.12 [conv.mem]); or

2. Add the indicated text to the note in the second bullet of 17.3.2 [temp.arg.nontype] paragraph 5:

   [*Note:* In particular, neither the null pointer conversion (7.11 [conv.ptr]) nor the derived-to-base conversion (7.11 [conv.ptr]) are applied. Although `0` is a valid *template-argument* for a non-type *template-parameter* of integral type, it is not a valid *template-argument* for a non-type *template-parameter* of pointer type. **However,** `(int*)0` **is a valid *template-argument* for a non-type *template-parameter* of type "pointer to int."** *—end note*]

3. Replace the normative wording of 17.5 [temp.type] paragraph 1 with the following:

   Two *template-id*s refer to the same class or function if

   - their *template-name*s refer to the same template, and
   - their corresponding type *template-argument*s are the same type, and
   - their corresponding non-type *template-argument*s of integral or enumeration type have identical values, and
   - their corresponding non-type *template-argument*s of pointer type refer to the same external object or function or are both the null pointer value, and
   - their corresponding non-type *template-argument*s of pointer-to-member type refer to the same class member or are both the null member pointer value, and
   - their corresponding non-type *template-argument*ss for template parameters of reference type refer to the same external object or function, and
   - their corresponding template *template-argument*s refer to the same template.

---

603. Type equivalence and unsigned overflow

Section: 17.5 [temp.type] **Status:** CD1 **Submitter:** James Widman **Date:** 3 November 2006

[Voted into WP at April, 2007 meeting as part of paper N2258.]

One of the requirements for two *template-id*s to refer to the same class or function (17.5 [temp.type] paragraph 1) is that

- their corresponding non-type *template-argument*s of integral or enumeration type have identical values

If we have some template of the form

```
template <unsigned char c> struct A;
```

does this imply that `A<'\001'>` and `A<257>` (for an eight-bit `char`) refer to different specializations?

Jens Maurer: Looks like it should say something like, "their corresponding converted non-type template arguments of integral or enumeration type have identical values."

Proposed resolution (April, 2007):

The change to 17.5 [temp.type] paragraph 1 shown in document J16/07-0118 = WG21 N2258, in which the syntactic non-terminal *template-argument* is changed to the English term "template argument" is sufficient to remove the confusion about whether the value before or after conversion is used in matching *template-id*s.

---

679. Equivalence of *template-id*s and operator function templates

Section: 17.5 [temp.type] **Status:** CD1 **Submitter:** Richard Corden **Date:** 1 March, 2008

[Voted into the WP at the September, 2008 meeting.]

In order for two *template-id*s to refer to the same function, 17.5 [temp.type] paragraph 1, bullet 1 requires that

- their *template-name*s refer to the same template

This makes it impossible for two *template-id*s referring to operator function templates to be equivalent, because only *simple-template-id*s have a *template-name*, and a *template-id* referring to an operator function template is not a *simple-template-id* (17.2 [temp.names] paragraph 1).

Suggested resolution:

Change 17.5 [temp.type] paragraph 1, bullet 1 to read,

- their *template-name*s **or *operator-function-id*s** refer to the same template

Proposed resolution (June, 2008):

Change 17.5 [temp.type] paragraph 1, first bullet, as follows:

- their *template-name*s **or *operator-function-id*s** refer to the same template, and

---

582. Template conversion functions

Section: 17.6.2 [temp.mem] **Status:** CD1 **Submitter:** PremAnand Rao **Date:** 23 May 2006

[Voted into WP at April, 2007 meeting.]

The wholesale replacement of the phrase "template function" by the resolution of issue 105 seems to have overlooked the similar phrase "template conversion function." This phrase appears a number of times in 16.3.3.1.2 [over.ics.user] paragraph 3, 17.6.2 [temp.mem] paragraphs 5-8, and 17.9.2 [temp.deduct] paragraph 1. It should be systematically replaced in similar fashion to the resolution of issue 105.

Proposed resolution (October, 2006):

1. Change 16.3.3.1.2 [over.ics.user] paragraph 3 as follows:

   If the user-defined conversion is specified by a ~~template conversion function~~ **specialization of a conversion function template**, the second standard conversion sequence must have exact match rank.

2. Change 17.6.2 [temp.mem] paragraph 5 as follows:

   A specialization of a ~~template conversion function~~ **conversion function template** is referenced in the same way as a non-template conversion function that converts to the same type.

3. Change 17.6.2 [temp.mem] paragraph 6 as follows:

   A specialization of a ~~template conversion function~~ **conversion function template** is not found by name lookup. Instead, any ~~template conversion functions~~ **conversion function templates** visible in the context of the use are considered.

4. Change 17.6.2 [temp.mem] paragraph 7 as follows:

   A ~~using declaration~~ ***using-declaration*** in a derived class cannot refer to a specialization of a ~~template conversion function~~ **conversion function template** in a base class.

5. Change 17.6.2 [temp.mem] paragraph 8 as follows:

   Overload resolution (16.3.3.2 [over.ics.rank]) and partial ordering (17.6.6.2 [temp.func.order]) are used to select the best conversion function among multiple ~~template conversion functions~~ **specializations of conversion function templates** and/or non-template conversion functions.

6. Change 17.9.2.3 [temp.deduct.conv] paragraph 1 as follows:

   Template argument deduction is done by comparing the return type of the ~~template conversion function~~ **conversion function template** (call it `P`) with the type that is required as the result of the conversion (call it `A`) as described in 17.9.2.5 [temp.deduct.type].

---

329. Evaluation of friends of templates

Section: 17.6.4 [temp.friend] **Status:** CD1 **Submitter:** John Spicer **Date:** 19 Dec 2001

[Voted into WP at October 2003 meeting.]

17.6.4 [temp.friend] paragraph 5 says:

> When a function is defined in a friend function declaration in a class template, the function is defined at each instantiation of the class template. The function is defined even if it is never used. The same restrictions on multiple declarations and definitions which apply to non-template function declarations and definitions also apply to these implicit definitions. [Note: if the function definition is ill-formed for a given specialization of the enclosing class template, the program is ill-formed even if the function is never used.]

This means that the following program is invalid, even without the call of `f(ai)`:

```
template <class T> struct A {
  friend void f(A a) {
    g(a);
  }
};
int main()
{
  A<int> ai;
// f(ai);  // Error if f(ai) is actually called
}
```

The EDG front end issues an error on this case even if `f(ai)` is never called. Of the compilers I tried (g++, Sun, Microsoft, Borland) we are the only ones to issue such an error.

This issue came up because there is a library that either deliberately or accidentally makes use of friend functions that are not valid for certain instantiations.

The wording in the standard is the result of a deliberate decision made long ago, but given the fact that most implementations do otherwise it raises the issue of whether we did the right thing.

Upon further investigation, the current rule was adopted as the resolution to issue 6.47 in my series of template issue papers. At the time the issue was discussed (7/96) most compilers did evaluate such friends. So it seems that a number of compilers have changed their behavior since then.

Based on current practice, I think the standard should be changed to evaluate such friends only when used.

Proposed resolution (October 2002):

Change section 17.6.4 [temp.friend] paragraph 5 from:

> When a function is defined in a friend function declaration in a class template, the function is defined at each instantiation of the class template. The function is defined even if it is never used. The same restrictions on multiple declarations and definitions which apply to non-template function declarations and definitions also apply to these implicit definitions. [*Note:* if the function definition is ill-formed for a given specialization of the enclosing class template, the program is ill-formed even if the function is never used.]

to:

> When a function is defined in a friend function declaration in a class template, the function is instantiated when the function is used. The same restrictions on multiple declarations and definitions that apply to non-template function declarations and definitions also apply to these implicit definitions.

Note the change from "which" to "that" in the last sentence.

---

410. Paragraph missed in changes for issue 166

Section: 17.6.4 [temp.friend] **Status:** CD1 **Submitter:** John Spicer **Date:** 18 Apr 2003

[Voted into WP at October 2004 meeting.]

17.6.4 [temp.friend] paragraph 2 was overlooked when the changes for issue 166 were made.

The friend declaration of f<>(int) is now valid.

> A friend function declaration that is not a template declaration and in which the name of the friend is an unqualified *template-id* shall refer to a specialization of a function template declared in the nearest enclosing namespace scope. [*Example:*

```
namespace N {
        template <class T> void f(T);
        void g(int);
        namespace M {
                template <class T> void h(T);
                template <class T> void i(T);
                struct A {
                        friend void f<>(int);   // ill-formed - N::f
                        friend void h<>(int);   // OK - M::h
                        friend void g(int);     // OK - new decl of M::g
                        template <class T> void i(T);
                        friend void i<>(int);   // ill-formed - A::i
                };
        }
}
```

> *--end example*]

Proposed Resolution (October 2003):

Remove 17.6.4 [temp.friend] paragraph 2:

> A friend function declaration that is not a template declaration and in which the name of the friend is an unqualified *template-id* shall refer to a specialization of a function template declared in the nearest enclosing namespace scope. [*Example:*

```
namespace N {
        template <class T> void f(T);
        void g(int);
        namespace M {
                template <class T> void h(T);
                template <class T> void i(T);
                struct A {
                        friend void f<>(int);   // ill-formed - N::f
                        friend void h<>(int);   // OK - M::h
                        friend void g(int);     // OK - new decl of M::g
                        template <class T> void i(T);
                        friend void i<>(int);   // ill-formed - A::i
                };
```

```
        }
    }
```

--end example]

---

286. Incorrect example in partial specialization

Section: 17.6.5 [temp.class.spec] **Status:** CD1 **Submitter:** Martin Sebor **Date:** 09 May 2001

[Moved to DR at 4/02 meeting.]

The example in 17.6.5 [temp.class.spec] paragraph 6 is incorrect. It reads,

```
template<class T> struct A {
    class C {
        template<class T2> struct B { };
    };
};

// partial specialization of A<T>::C::B<T2>
template<class T> template<class T2>
    struct A<T>::C::B<T2*> { };

A<short>::C::B<int*> absip; // uses partial specialization
```

Because `C` is a `class` rather than a `struct`, the use of the name `B` is inaccessible.

Proposed Resolution (10/01):

Change `class C` to `struct C` in the example in 17.6.5 [temp.class.spec] paragraph 6. The example becomes

```
template<class T> struct A {
    struct C {
        template<class T2> struct B { };
    };
};

// partial specialization of A<T>::C::B<T2>
template<class T> template<class T2>
    struct A<T>::C::B<T2*> { };

A<short>::C::B<int*> absip; // uses partial specialization
```

---

517. Partial specialization following explicit instantiation

Section: 17.6.5 [temp.class.spec] **Status:** CD1 **Submitter:** John Spicer **Date:** 03 May 2005

[Voted into WP at the October, 2006 meeting.]

According to 17.6.5 [temp.class.spec] paragraph 1,

> If a template is partially specialized then that partial specialization shall be declared before the first use of that partial specialization that would cause an implicit instantiation to take place, in every translation unit in which such a use occurs; no diagnostic is required.

This leaves the impression that an explicit instantiation of the primary template may precede the declaration of an applicable partial specialization. Is the following example well-formed?

```
template<typename T> class X{
    public:
    void foo(){};
};

template class X<void *>;

template<typename T> class X<T*>{
    public:
    void baz();
};

void bar() {
    X<void *> x;
    x.foo();
}
```

Proposed resolution (October, 2005):

Replace the last sentence of 17.6.5 [temp.class.spec] paragraph 1:

> If a template is partially specialized then that partial specialization shall be declared before the first use of that partial specialization that would cause an implicit instantiation to take place, in every translation unit in which such a use occurs; no diagnostic is required.

with:

> A partial specialization shall be declared before the first use of a class template specialization that would make use of the partial specialization as the result of an implicit or explicit instantiation in every translation unit in which such a use occurs; no diagnostic is required.

---

214. Partial ordering of function templates is underspecified

Section: 17.6.6.2 [temp.func.order] **Status:** CD1 **Submitter:** Martin von Loewis/Martin Sebor **Date:** 13 Mar 2000

[Voted into WP at October 2003 meeting.]

In 17.6.6.2 [temp.func.order], partial ordering is explained in terms of template argument deduction. However, the exact procedure for doing so is not specified. A number of details are missing, they are explained as sub-issues below.

1. 17.6.6.2 [temp.func.order] paragraph 2 refers to 17.9.2 [temp.deduct] for argument deduction. This is the wrong reference; it explains how explicit arguments are processed (paragraph 2) and how function parameter types are adjusted (paragraph 3). Neither of these steps is meaningful in the context of partial ordering. Next in deduction follows one of the steps in 17.9.2.1 [temp.deduct.call], 17.9.2.2 [temp.deduct.funcaddr], 17.9.2.3 [temp.deduct.conv], or 17.9.2.5 [temp.deduct.type]. The standard does not specify which of these contexts apply to partial ordering.
2. Because 17.9.2.1 [temp.deduct.call] and 17.9.2.3 [temp.deduct.conv] both start with actual function parameters, it is meaningful to assume that partial ordering uses 17.9.2.5 [temp.deduct.type], which only requires types. With that assumption, the example in 17.6.6.2 [temp.func.order] paragraph 5 becomes incorrect, considering the two templates

   ```
   template<class T> void g(T);  // #1
   template<class T> void g(T&); // #2
   ```

   Here, #2 is at least as specialized as #1: With a synthetic type `U`, #2 becomes `g(U&)`; argument deduction against #1 succeeds with `T=U&`. However, #1 is not at least as specialized as #2: Deducing `g(U)` against `g(T&)` fails. Therefore, the second template is more specialized than the first, and the call `g(x)` is not ambiguous.
3. According to John Spicer, the intent of the partial ordering was that it uses deduction as in a function call (17.9.2.1 [temp.deduct.call]), which is indicated by the mentioning of "exact match" in 17.6.6.2 [temp.func.order] paragraph 4. If that is indeed the intent, it should be specified how values are obtained for the step in 17.9.2.1 [temp.deduct.call] paragraph 1, where the types of the arguments are determined. Also, since 17.9.2.1 [temp.deduct.call] paragraph 2 drops references from the parameter type, symmetrically, references should be dropped from the argument type (which is done in 8 [expr] paragraph 2, for a true function call).
4. 17.6.6.2 [temp.func.order] paragraph 4 requires an "exact match" for the "deduced parameter types". It is not clear whether this refers to the template parameters, or the parameters of the template function. Considering the example

   ```
   template<class S> void g(S);  // #1
   template<class T> void g(T const &); // #3
   ```

   Here, #3 is clearly at least as specialized as #1. To determine whether #1 is at least as specialized as #3, a unique type `U` is synthesized, and deduction of `g<U>(U)` is performed against #3. Following the rules in 17.9.2.1 [temp.deduct.call], deduction succeeds with `T=U`. Since the template argument is `U`, and the deduced template parameter is also `U`, we have an exact match between the template parameters. Even though the conversion from `U` to `U const &` is an exact match, it is not clear whether the added qualification should be taken into account, as it is in other places.

Issue 200 covers a related issue, illustrated by the following example:

```
template <class T> T f(int);
template <class T, class U> T f(U);
void g() {
    f<int>(1);
}
```

Even though one template is "obviously" more specialized than the other, deduction fails in both directions because neither function parameter list allows template parameter `T` to be deduced.

(See also issue 250.)

Nathan Sidwell:

17.6.6.2 [temp.func.order] describes the partial ordering of function templates. Paragraph 5 states,

> A template is more specialized than another if, and only if, it is at least as specialized as the other template and that template is not at least as specialized as the first.

To paraphrase, given two templates A & B, if A's template parameters can be deduced by B, but B's cannot be deduced by A, then A is more specialized than B. Deduction is done as if for a function call. In particular, except for conversion operators, the return type is not involved in deduction. This leads to the following templates and use being unordered. (This example is culled from G++ bug report 4672 http://gcc.gnu.org/cgi-bin/gnatsweb.pl?cmd=view&pr=4672)

```
template <typename T, class U> T checked_cast(U from); //#1
template <typename T, class U> T checked_cast(U * from); //#2
class C {};

void foo (int *arg)
{
  checked_cast <C const *> (arg);
}
```

In the call,

#1 can be deduced with T = 'C const *' and U = 'int *'

#2 can be deduced with T = 'C const *' and U = 'int'

It looks like #2 is more specialized that #1, but 17.6.6.2 [temp.func.order] does not make it so, as neither template can deduce 'T' from the other template's function parameters.

Possible Resolutions:

There are several possible solutions, however through experimentation I have discounted two of them.

Option 1:

When deducing function ordering, if the return type of one of the templates uses a template parameter, then return types should be used for deduction. This, unfortunately, makes existing well formed programs ill formed. For example

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

template <class T> X<T> Foo (T *);    // #1
template <class T> int Foo (T const *); // #2

void Baz (int *p1, int const *p2)
{
  int j = Foo (p2); //#3
}
```

Here, neither #1 nor #2 can deduce the other, as the return types fail to match. Considering only the function parameters gives #2 more specialized than #1, and hence makes the call #3 well formed.

Option 2:

As option 1, but only consider the return type when deducing the template whose return type involves template parameters. This has the same flaw as option 1, and that example is similarly ill formed, as #1's return type 'X<T,0>' fails to match 'int' so #1 cannot deduce #2. In the converse direction, return types are not considered, but the function parameters fail to deduce.

Option 3:

It is observed that the original example is only callable with a template-id-expr to supply a value for the first, undeducible, parameter. If that parameter were deducible it would also appear within at least one of the function parameters. We can alter paragraph 4 of [temp.func.order] to indicate that it is not necessary to deduce the parameters which are provided explicitly, when the call has the form of a template-id-expr. This is a safe extension as it only serves to make ill formed programs well formed. It is also in line with the concept that deduction for function specialization order should proceed in a similar manner to function calling, in that explicitly provided parameter values are taken into consideration.

Suggested resolution:

Insert after the first sentence of paragraph 4 in 17.6.6.2 [temp.func.order]

> Should any template parameters remain undeduced, and the function call be of the form of a *template-id-expr*, those template parameters provided in the *template-id-expr* may be arbitrarily synthesized prior to determining whether the deduced arguments generate a valid function type.

See also issue 200.

(April 2002) John Spicer and John Wiegley have written a paper on this. See 02-0051/N1393.

Proposed resolution (October 2002):

Change 17.6.6.2 [temp.func.order] paragraph 2 to read:

> Partial ordering selects which of two function templates is more specialized than the other by transforming each template in turn (see next paragraph) and performing template argument deduction using the function parameter types, or in the case of a conversion function the return type. The deduction process determines whether one of the templates is more specialized than the other. If so, the more specialized template is the one chosen by the partial ordering process.

Change 17.6.6.2 [temp.func.order] paragraph 3 to read:

> To produce the transformed template, for each type, non-type, or template template parameter synthesize a unique type, value, or class template respectively and substitute it for each occurrence of that parameter in the function type of the template.

Change 17.6.6.2 [temp.func.order] paragraph 4 to read (note: the section reference should refer to the section added to 17.9.2 [temp.deduct] below):

> Using the transformed function template's function parameter list, or in the case of a conversion function its transformed return type, perform type deduction against the function parameter list (or return type) of the other function. The mechanism for performing these deductions is given in 14.8.2.x.

Remove the text of 17.6.6.2 [temp.func.order] paragraph 5 but retain the example. The removed text is:

A template is more specialized than another if, and only if, it is at least as specialized as the other template and that template is not at least as specialized as the first.

Insert the following section before 14.8.2.5 (Note that this would either be a new 14.8.2.4, or would be given a number like 14.8.2.3a. If neither of these is possible from a troff point of view, this could be made 14.8.2.5.)

Deducing template arguments when determining the partial ordering of function templates (temp.deduct.partial)

Template argument deduction is done by comparing certain types associated with the two function templates being compared.

Two sets of types are used to determine the partial ordering. For each of the templates involved there is the original function type and the transformed function type. [*Note:* The creation of the transformed type is described in 17.6.6.2 [temp.func.order].] The deduction process uses the transformed type as the argument template and the original type of the other template as the parameter template. This process is done twice for each type involved in the partial ordering comparison: once using the transformed template-1 as the argument template and template-2 as the parameter template and again using the transformed template-2 as the argument template and template-1 as the parameter template.

The types used to determine the ordering depend on the context in which the partial ordering is

- In the context of a function call, the function parameter types are used.
- In the context of a call to a conversion operator, the return types of the conversion function templates are used.
- In other contexts (17.6.6.2 [temp.func.order]), the function template's function type is used.

Each type from the parameter template and the corresponding type from the argument template are used as the types of `P` and `A`.

Before the partial ordering is done, certain transformations are performed on the types used for partial ordering:

- If `P` is a reference type, `P` is replaced by the type referred to.
- If `A` is a reference type, `A` is replaced by the type referred to.

If both `P` and `A` were reference types (before being replaced with the type referred to above), determine which of the two types (if any) is more cv-qualified than the other; otherwise the types are considered to be equally cv-qualified for partial ordering purposes. The result of this determination will be used below.

Remove any top-level cv-qualifiers:

- If `P` is a cv-qualified type, `P` is replaced by the cv-unqualified version of `P`.
- If `A` is a cv-qualified type, `A` is replaced by the cv-unqualified version of `A`.

Using the resulting types `P` and `A` the deduction is then done as described in (17.9.2.5 [temp.deduct.type]). If deduction succeeds for a given type, the type from the argument template is considered to be at least as specialized as the type from the parameter template.

If, for a given type, deduction succeeds in both directions (i.e., the types are identical after the transformations above) if the type from the argument template is more cv-qualified than the type from the parameter template (as described above) that type is considered to be more specialized than the other. If neither type is more cv-qualified than the other then neither type is more specialized than the other.

If for each type being considered a given template is at least as specialized for all types and more specialized for some set of types and the other template is not more specialized for any types or is not at least as specialized for any types, then the given template is more specialized than the other template. Otherwise, neither template is more specialized than the other.

In most cases, all template parameters must have values in order for deduction to succeed, but for partial ordering purposes a template parameter may remain without a value provided it is not used in the types being used for partial ordering. [*Note:* A template parameter used in a non-deduced context is considered used.]

[*Example:*

```
template <class T> T f(int);        // #1
template <class T, class U> T f(U); // #2
void g() {
    f<int>(1);  // Calls #1
}
```

--end example]

---

180. `typename` and elaborated types

Section: 17.7 [temp.res] **Status:** CD1 **Submitter:** Mike Miller **Date:** 21 Dec 1999

[Moved to DR at 4/02 meeting.]

Mike Miller: A question about `typename` came up in the discussion of issue 68 that is somewhat relevant to the idea of omitting typename in contexts where it is clear that a type is required: consider something like

```
template <class T>
class X {
```

```
        friend class T::nested;
    };
```

Is `typename` required here? If so, where would it go? (The grammar doesn't seem to allow it anywhere in an *elaborated-type-specifier* that has a *class-key*.)

Bill Gibbons: The `class` applies to the last identifier in the qualified name, since all the previous names must be classes or namespaces. Since the name is specified to be a class it does not need `typename`. [However,] it looks like 17.7 [temp.res] paragraph 3 requires `typename` and the following paragraphs do not exempt this case. This is not what we agreed on.

Proposed resolution (04/01):

In 17.7 [temp.res] paragraph 5, change

> The keyword `typename` is not permitted in a *base-specifier* or in a *mem-initializer*, in these contexts a *qualified-name* that depends on a *template-parameter* (17.7.2 [temp.dep]) is implicitly assumed to be a type name.

to

> A qualified name used as the name in a *mem-initializer-id*, a *base-specifier*, or an *elaborated-type-specifier* (in the *class-key* and `enum` forms) is implicitly assumed to name a type, without the use of the `typename` keyword. [*Note:* the `typename` keyword is not permitted by the syntax of these constructs.]

(The expected resolution for issue 254 will remove the `typename` forms from the grammar for *elaborated-type-specifier*. If that resolution is adopted, the parenthetical phrase "(in the *class-key* and `enum` forms)" in the preceding wording should be removed because those will be the only forms of *elaborated-type-specifier*.)

This has been consolidated with the edits for some other issues. See N1376=02-0034.

---

345. Misleading comment on example in templates chapter

Section: 17.7 [temp.res] **Status:** CD1 **Submitter:** Jason Shirk **Date:** 18 March 2002

[Voted into WP at April 2003 meeting.]

The following example from 17.7 [temp.res] paragraph 4:

```
struct A {
        struct X { };
        int X;
};
template<class T> void f(T t) {
        typename T::X x;        //  ill-formed: finds the data member  X
                                //  not the member type  X
}
```

is not ill-formed. The intent of the example is obvious, but some mention should be made that it is only ill-formed when T=A. For other T's, it could be well formed.

Proposed resolution (October 2002):

In 17.7 [temp.res] paragraph 4, replace the example with:

```
struct A {
  struct X { };
  int X;
} ;
struct B {
  struct X { };
} ;
template<class T> void f(T t) {
  typename T::X x;
}
void foo() {
  A a;
  B b;
  f(b);  // OK -- T::X refers to B::X.
  f(a);  // error: T::X refers to the data member A::X not
         // the struct A::X.
}
```

---

382. Allow `typename` outside of templates

Section: 17.7 [temp.res] **Status:** CD1 **Submitter:** Steve Adamczyk **Date:** 8 Nov 2002

[Voted into WP at October 2005 meeting.]

P. J. Plauger, among others, has noted that `typename` is hard to use, because in a given context it's either required or forbidden, and it's often hard to tell which. It would make life easier for programmers if `typename` could be allowed in places where it is not required, e.g., outside of templates.

Notes from the April 2003 meeting:

There was unanimity on relaxing this requirement on typename. The question was how much to relax it. Everyone agreed on allowing it on all qualified names, which is an easy fix (no syntax change required). But should it be allowed other places? P.J. Plauger said he'd like to see it allowed anywhere a type name is allowed, and that it could actually be a decades-late assist for the infamous "the ice is thin here" typedef problem noted in K&R I.

Proposed resolution (April 2003):

Replace the text at the start of 17.7 [temp.res] paragraph 3:

> A *qualified-id* that refers to a type and in which the *nested-name-specifier* depends on a *template-parameter* (17.7.2 [temp.dep]) shall be prefixed by the keyword `typename` to indicate that the *qualified-id* denotes a type, forming an *elaborated-type-specifier* (10.1.7.3 [dcl.type.elab]).

With:

> The keyword `typename` can only be applied to a *qualified-id*. A *qualified-id* that refers to a type and in which the *nested-name-specifier* depends on a *template-parameter* (17.7.2 [temp.dep]) shall be prefixed by the keyword `typename` to indicate that the *qualified-id* denotes a type, forming an *elaborated-type-specifier* (10.1.7.3 [dcl.type.elab]). If a *qualified-id* which has been prefixed by the keyword `typename` does not denote a type the program is ill-formed. [*Note:* The keyword is only required on a *qualified-id* within a template declaration or definition in which the *nested-name-specifier* depends on a *template-parameter*.]

Remove 17.7 [temp.res] paragraph 5:

> The keyword `typename` shall only be used in template declarations and definitions, including in the return type of a function template or member function template, in the return type for the definition of a member function of a class template or of a class nested within a class template, and in the *type-specifier* for the definition of a static member of a class template or of a class nested within a class template. The keyword `typename` shall be applied only to qualified names, but those names need not be dependent. The keyword `typename` shall be used only in contexts in which dependent names can be used. This includes template declarations and definitions but excludes explicit specialization declarations and explicit instantiation declarations. The keyword `typename` is not permitted in a *base-specifier* or in a *mem-initializer*, in these contexts a *qualified-id* that depends on a *template-parameter* (temp.dep) is implicitly assumed to be a type name.

Note: the claim here that a qualified name preceded by `typename` forms an elaborated type specifier conflicts with the changes made in issue 254 (see N1376=02-0034), which introduces *typename-specifier*.

Notes from October 2003 meeting:

We considered whether `typename` should be allowed in more places, and decided we only wanted to allow it in qualified names (for now at least).

Core issue 254 changed *elaborated-type-specifier* to *typename-specifier*. It also changed 17.7 [temp.res] paragraph 5, which this proposed resolution deletes.

See also issue 468.

Proposed resolution (October, 2004):

1. Change 17.7 [temp.res] paragraph 3 as follows:

> ~~A~~ **When a** *qualified-id* ~~that refers to a type and~~ in which the *nested-name-specifier* depends on a *template-parameter* (17.7.2 [temp.dep]) **is intended to refer to a type, it** shall be prefixed by the keyword `typename` ~~to indicate that the qualified-id denotes a type~~, forming a *typename-specifier*. **If the *qualified-id* in a *typename-specifier* does not denote a type, the program is ill-formed.**

2. Change 17.7 [temp.res] paragraph 5 as follows:

> ~~The keyword `typename` shall only be used in template declarations and definitions, including in the return type of a function template or member function template, in the return type for the definition of a member function of a class template or of a class nested within a class template, and in the *type-specifier* for the definition of a static member of a class template or of a class nested within a class template. The keyword `typename` shall be applied only to qualified names, but those names need not be dependent. The keyword `typename` shall be used only in contexts in which dependent names can be used. This includes template declarations and definitions but excludes explicit specialization declarations and explicit instantiation declarations.~~ A qualified name used as the name in a *mem-initializer-id*, a *base-specifier*, or an *elaborated-type-specifier* is implicitly assumed to name a type, without the use of the `typename` keyword. [*Note:* the `typename` keyword is not permitted by the syntax of these constructs. —*end note*]

---

409. Obsolete paragraph missed by changes for issue 224

Section: 17.7 [temp.res] **Status:** CD1 **Submitter:** John Spicer **Date:** 18 Apr 2003

[Voted into WP at October 2004 meeting.]

Paragraph 6 of 17.7 [temp.res] is obsolete as result of issue 224, and needs to be revised.

Within the definition of a class template or within the definition of a member of a class template, the keyword typename is not required when referring to the unqualified name of a previously declared member of the class template that declares a type. The keyword typename shall always be specified when the member is referred to using a qual- ified name, even if the qualifier is simply the class template name. [*Example:*

```
template<class T> struct A {
    typedef int B;
    A::B b;                     // ill-formed: typename required before A::B
    void f(A<T>::B);            // ill-formed: typename required before A<T>::B
    typename A::B g();          // OK
};
```

]

Proposed Resolution:

Change 17.7 [temp.res] paragraph 6 as follows

Within the definition of a class template or within the definition of a member of a class template, the keyword `typename` is not required when referring to the unqualified name of a previously declared member of the class template that declares a type. ~~The keyword `typename` shall always be specified when the member is referred to using a qualified name, even if the qualifier is simply the class template name.~~ [Example:

```
template<class T> struct A {
    typedef int B;
    B b;                        // ok, no typename required
    A::B b;                     // ill-formed: typename required before A::B
    void f(A<T>::B);            // ill-formed: typename required before A<T>::B
    typename A::B g();          // OK
};
```

~~The keyword typename is required whether the qualified name is A or A<T> because A or A<T> are synonyms within a class template with the parameter list <T>.~~]

---

559. Editing error in issue 382 resolution

Section: 17.7 [temp.res] **Status:** CD1 **Submitter:** Mike Miller **Date:** 11 February 2006

[Voted into WP at April, 2006 meeting.]

Part of the resolution for issue 224 was the addition of the phrase "but does not refer to a member of the current instantiation" to 17.7 [temp.res] paragraph 3. When the resolution of issue 382 was added to the current working draft, however, that phrase was inadvertently removed. Equivalent phrasing should be restored.

Proposed resolution (April, 2006):

Replace the first sentence of 17.7 [temp.res] paragraph 3 with the following text:

When a *qualified-id* is intended to refer to a type that is not a member of the current instantiation (17.7.2.1 [temp.dep.type]) and its *nested-name-specifier* depends on a *template-parameter* (17.7.2 [temp.dep]), it shall be prefixed by the keyword `typename`, forming a *typename-specifier*.

---

666. Dependent *qualified-id*s without the `typename` keyword

Section: 17.7 [temp.res] **Status:** CD1 **Submitter:** Daveed Vandevoorde **Date:** 6 December 2007

[Voted into the WP at the June, 2008 meeting.]

17.7 [temp.res] paragraphs 2 and 4 read,

A name used in a template declaration or definition and that is dependent on a *template-parameter* is assumed not to name a type unless the applicable name lookup finds a type name or the name is qualified by the keyword *typename*.

If a specialization of a template is instantiated for a set of *template-argument*s such that the *qualified-id* prefixed by `typename` does not denote a type, the specialization is ill-formed.

It is not clear whether this is intended to, or is sufficient to, render a specialization ill-formed if a dependent *qualified-id* that is *not* prefixed by `typename` actually *does* denote a type. For example,

```
int i;

template <class T> void f() {
    T::x * i; // declaration or multiplication!?
}

struct Foo {
    typedef int x;
};
```

```
struct Bar {
    static int const x = 5;
};

int main() {
    f<Bar>(); // multiplication
    f<Foo>(); // declaration!
}
```

I think that the specialization for `Foo` should be ill-formed.

Proposed resolution (February, 2008):

Add the following after 17.7 [temp.res] paragraph 5:

> If, for a given set of template arguments, a specialization of a template is instantiated that refers to a *qualified-id* that denotes a type, and the *nested-name-specifier* of the *qualified-id* depends on a template parameter, the *qualified-id* shall either be prefixed by `typename` or shall be used in a context in which it implicitly names a type as described above. [*Example:*
>
> ```
> template <class T> void f(int i) {
>   T::x * i;     // T::x must not be a type
> }
>
> struct Foo {
>   typedef int x;
> };
>
> struct Bar {
>   static int const x = 5;
> };
>
> int main() {
>   f<Bar>(1);    // OK
>   f<Foo>(1);    // error: Foo::x is a type
> }
> ```
>
> —*end example*]

---

515. Non-dependent references to base class members

Section: 17.7.2 [temp.dep] **Status:** CD1 **Submitter:** Mike Miller **Date:** 18 Apr 2005

[Voted into WP at the October, 2006 meeting.]

Implementations vary in their treatment of the following code:

```
struct A {
  int foo_;
};
template <typename T> struct B: public A { };
template <typename T> struct C: B<T> {
  int foo() {
    return A::foo_;  // #1
  }
};
int f(C<int>* p) {
  return p->foo();
}
```

According to one analysis, because the expression `A::foo_` on line #1 is non-dependent, it must be analyzed in the definition context. It that context, it violates the restrictions of 12.2 [class.mem] paragraph 10 on how the name of a nonstatic data member of a class can be used and thus should be treated as an error.

On the other hand, the description of the transformation of an *id-expression* into a class member access expression (12.2.2 [class.mfct.non-static] paragraph 3) does not have any special treatment of templates; when `C<int>::foo()` is instantiated, the reference to `A::foo_` turns out to be to a base class member and is thus transformed into `(*this).A::foo_` and is thus not an error.

Proposed resolution (October, 2005):

Change 12.2.2 [class.mfct.non-static] paragraph 3 as indicated:

> When an *id-expression* (_N4567_.5.1.1 [expr.prim.general]) that is not part of a class member access syntax (8.2.5 [expr.ref]) and not used to form a pointer to member (8.3.1 [expr.unary.op]) is used in the body of a non-static member function of class `X` or used in the *mem-initializer* for a constructor of class `X`, if name lookup (6.4.1 [basic.lookup.unqual]) resolves the name in the *id-expression* to a non-static non-type member of ~~class `X` or of a base class of `X`~~ **some class** `C`, the *id-expression* is transformed into a class member access expression (8.2.5 [expr.ref]) using `(*this)` (12.2.2.1 [class.this]) as the *postfix-expression* to the left of the `.` operator. **[*Note:* If `C` is not `X` or a base class of `X`, the class member access expression is ill-formed. —*end note*]** ~~The member name then refers to the member of the object for which the function is called.~~ Similarly during name lookup...

---

524. Can function-notation calls to operator functions be dependent?

Section: 17.7.2 [temp.dep] **Status:** CD1 **Submitter:** Daveed Vandevoorde **Date:** 19 July 2005

[Voted into WP at the October, 2006 meeting.]

The description of dependent function calls in 17.7.2 [temp.dep] paragraph 1 applies only to *identifier*s in postfix-notation function calls and to operator notation calls for operator functions:

> In an expression of the form:
>
> *postfix-expression* (*expression-list*$_{opt}$)
>
> where the *postfix-expression* is an *identifier*, the *identifier* denotes a *dependent name* if and only if any of the expressions in the *expression-list* is a type-dependent expression (17.7.2.2 [temp.dep.expr]). If an operand of an operator is a type-dependent expression, the operator also denotes a dependent name.

It would appear from the related passage in 17.7.4.2 [temp.dep.candidate] paragraph 1 that the description of postfix-notation function calls should apply to all *unqualified-id*s that are not *template-id*s, including *operator-function-id*s, not just to *identifier*s:

> For a function call that depends on a template parameter, if the function name is an *unqualified-id* but not a *template-id*, the candidate functions are found...

Proposed resolution (October, 2005):

1. Change 17.7.2 [temp.dep] paragraph 1 as indicated:

   > ...In an expression of the form:
   >
   > *postfix-expression* (*expression-list*$_{opt}$)
   >
   > where the *postfix-expression* is an ~~*identifier*~~ ***unqualified-id*** **but not a** ***template-id***, the ~~*identifier*~~ ***unqualified-id*** denotes a *dependent name* if and only if any of the expressions in the *expression-list* is a type-dependent expression (17.7.2.2 [temp.dep.expr])...

2. Change 17.7.4.2 [temp.dep.candidate] paragraph 1 as indicated:

   > For a function call that depends on a template parameter, if the function name is an *unqualified-id* but not a *template-id*, **or if the function is called using operator notation,** the candidate functions are found using the usual lookup rules (6.4.1 [basic.lookup.unqual], 6.4.2 [basic.lookup.argdep]) except that...

(See also issue 561.)

---

224. Definition of dependent names

Section: 17.7.2.1 [temp.dep.type] **Status:** CD1 **Submitter:** Derek Inglis **Date:** 30 Nov 1999

[Moved to DR at 10/01 meeting.]

The definition of when a type is dependent, given in 17.7.2.1 [temp.dep.type], is essentially syntactic: if the reference is a *qualified-id* and one of the *class-name*s in the *nested-name-specifier* is dependent, the type is dependent. This approach leads to surprising results:

```
template <class T> class X {
    typedef int I;
    I a;                // non-dependent
    typename X<T>::I b; // dependent
    typename X::I c;    // dependent (X is equivalent to X<T>)
};
```

Suggested resolution:

The decision on whether a name is dependent or non-dependent should be based on lookup, not on the form of the name: if the name can be looked up in the definition context and cannot be anything else as the result of specialization, the name should be non-dependent.

See papers J16/00-0028 = WG21 N1251 and J16/00-0056 = WG21 N1279.

Proposed resolution (10/00):

1. Replace section 17.7.2.1 [temp.dep.type] with the following:

   > In the definition of a class template, a nested class of a class template, a member of a class template, or a member of a nested class of a class template, a name refers to the *current instantiation* if it is
   >
   > - the injected-class-name (clause 12 [class]) of the class template or nested class,
   > - in the definition of a primary class template, the name of the class template followed by the template argument list of the primary template (as described below) enclosed in <>,
   > - in the definition of a nested class of a class template, the name of the nested class referenced as a member of the current instantiation, or

- in the definition of a partial specialization, the name of the class template followed by the template argument list of the partial specialization enclosed in <>.

The template argument list of a primary template is a template argument list in which the n^{th} template argument has the value of the n^{th} template parameter of the class template.

A template argument that is equivalent to a template parameter (i.e., has the same constant value or the same type as the template parameter) can be used in place of that template parameter in a reference to the current instantiation. In the case of a nontype template argument, the argument must have been given the value of the template parameter and not an expression involving the template parameter.

[*Example:*

```
template <class T> class A {
    A* p1;      // A is the current instantiation
    A<T>* p2;   // A<T> is the current instantiation
    A<T*> p3;   // A<T*> is not the current instantiation
    ::A<T>* p4; // ::A<T> is the current instantiation
    class B {
        B* p1;        // B is the current instantiation
        A<T>::B* p2;  // A<T>::B is the current instantiation
        typename A<T*>::B* p3; // A<T*>::B is not the
                               // current instantiation
    };
};

template <class T> class A<T*> {
    A<T*>* p1;  // A<T*> is the current instantiation
    A<T>* p2;   // A<T> is not the current instantiation
};

template <class T1, class T2, int I> struct B {
    B<T1, T2, I>*       b1;        // refers to the current instantiation
    B<T2, T1, I>*       b2;        // not the current instantiation
    typedef T1 my_T1;
    static const int my_I = I;
    static const int my_I2 = I+0;
    static const int my_I3 = my_I;
    B<my_T1, T2, my_I>* b3;  // refers to the current instantiation
    B<my_T1, T2, my_I2>* b4; // not the current instantiation
    B<my_T1, T2, my_I3>* b5; // refers to the current instantiation
};
```

—*end example*]

A name is a *member of the current instantiation* if it is

- An unqualified name that, when looked up, refers to a member of a class template. [*Note:* This can only occur when looking up a name in a scope enclosed by the definition of a class template.]
- A *qualified-id* in which the *nested-name-specifier* refers to the current instantiation.

[*Example:*

```
template <class T> class A {
    static const int i = 5;
    int n1[i];        // i refers to a member of the current instantiation
    int n2[A::i];     // A::i refers to a member of the current instantiation
    int n3[A<T>::i];  // A<T>::i refers to a member of the current instantiation
    int f();
};

template <class T> int A<T>::f()
{
    return i;  // i refers to a member of the current instantiation
}
```

—*end example*]

A name is a *member of an unknown specialization* if the name is a *qualified-id* in which the *nested-name-specifier* names a dependent type that is not the current instantiation.

A type is dependent if it is

- a template parameter,
- a member of an unknown specialization,
- a nested class that is a member of the current instantiation,
- a cv-qualified type where the cv-unqualified type is dependent,
- a compound type constructed from any dependent type,
- an array type constructed from any dependent type or whose size is specified by a constant expression that is value-dependent, or
- a *template-id* in which either the template name is a template parameter or any of the template arguments is a dependent type or an expression that is type-dependent or value-dependent.

[*Note:* Because typedefs to not introduce new types, but instead simply refer to other types, a name that refers to a typedef that is a member of the current instantiation is dependent only if the type referred to is dependent.]

2. In 17.7.2.2 [temp.dep.expr] paragraph 3, replace

- a *nested-name-specifier* that contains a *class-name* that names a dependent type.

with

- a *nested-name-specifier* or *qualified-id* that names a member of an unknown specialization.

3. In 17.7.2.2 [temp.dep.expr], add the following paragraph:

   A class member access expression (8.2.5 [expr.ref]) is type-dependent if the type of the referenced member is dependent. [*Note:* In an expression of the form `x.y` or `xp->y` the type of the expression is usually the type of the member `y` of the class of `x` (or the class pointed to by `xp`). However, if `x` or `xp` refers to a dependent type that is not the current instantiation, the type of `y` is always dependent. If `x` or `xp` refers to a non-dependent type or refers to the current instantiation, the type of `y` is the type of the class member access expression.]

4. In 17.7 [temp.res] paragraph 3, replace

   A *qualified-name* that refers to a type and that depends on a *template-parameter* (17.7.2 [temp.dep]) shall be prefixed by the keyword `typename`.

   with

   A *qualified-id* that refers to a type and that depends on a *template-parameter* (17.7.2 [temp.dep]) but does not refer to a member of the current instantiation shall be prefixed by the keyword `typename`.

   Note: the wording for this paragraph was changed in TC1. The words shown here are the pre-TC1 words.

5. In 17.2 [temp.names] paragraph 4, replace

   When the name of a member template specialization appears after `.` or `->` in a *postfix-expression*, or after a *nested-name-specifier* in a *qualified-id*, and the *postfix-expression* or *qualified-id* explicitly depends on a *template-parameter* (17.7.2 [temp.dep]), the member template name must be prefixed by the keyword `template`. Otherwise the name is assumed to name a non-template.

   with

   When the name of a member template specialization appears after `.` or `->` in a *postfix-expression*, or after a *nested-name-specifier* in a *qualified-id*, and the *postfix-expression* or *qualified-id* explicitly depends on a *template-parameter* (17.7.2 [temp.dep]) but does not refer to a member of the current instantiation (17.7.2.1 [temp.dep.type]), the member template name must be prefixed by the keyword `template`. Otherwise the name is assumed to name a non-template.

6. In 17.7.1 [temp.local] paragraph 2, remove the following text, which was added for issue 108. The updated definition of dependent name now addresses this case.

   Within the scope of a class template, when the unqualified name of a nested class of the class template is referred to, it is equivalent to the name of the nested class qualified by the name of the enclosing class template. [*Example:*

```
template <class T> struct A {
        class B {};
        // B is equivalent to A::B, which is equivalent to A<T>::B,
        // which is dependent.
        class C : B { };
};
```

   —*end example*]

---

447. Is offsetof type-dependent?

Section: 17.7.2.3 [temp.dep.constexpr] **Status:** CD1 **Submitter:** Mark Mitchell **Date:** 7 Jan 2004

[Voted into WP at October 2005 meeting.]

As far as I can tell, the standard doesn't say whether "offsetof(...)" is type-dependent. In the abstract, it shouldn't be -- an "offsetof" expression is always of type "size_t". But the standard doesn't say to what the definition of the macro is, so I don't think one can deduce that it will always be considered non-dependent by a conforming compiler.

John Spicer: (1) I agree that you can't know if offsetof is dependent because you don't know what it expands to. (2) In principle, offsetof should be like sizeof -- it is value-dependent if its argument is type-dependent.

Mark Mitchell: I think we should say that: (a) offsetof is not type-dependent, and (b) offsetof is value dependent iff the first argument is type-dependent

Everyone is using slightly different builtins to implement this functionality, and I don't think that there's any guarantee that they're all behaving the same here.

Notes from the March 2004 meeting:

Note that any such requirement would be in the library section, not core.

Proposed resolution (October, 2004):

1. At the end of 17.7.2.2 [temp.dep.expr] paragraph 4, add after the list that ends with `throw` *assignment-expression*:

[*Note:* For the standard library macro `offsetof`, see 21.2 [support.types]. —*end note*]

2. At the end of 17.7.2.3 [temp.dep.constexpr] paragraph 2, add after the list that ends with `sizeof(`*type-id*`)`:

   [*Note:* For the standard library macro `offsetof`, see 21.2 [support.types]. —*end note*]

3. In 21.2 [support.types] paragraph 4, replace

   The macro `offsetof` accepts a restricted set of *type* arguments in this International Standard. If *type* is not a POD structure or a POD union the results are undefined. The result of applying the `offsetof` macro to a field that is a static data member or a function member is undefined.

   with

   The macro `offsetof(`*type*, *member-designator*`)` accepts a restricted set of *type* arguments in this International Standard. If *type* is not a POD structure or a POD union (clause 12 [class]), the results are undefined. The expression `offsetof(`*type*, *member-designator*`)` is never type-dependent (17.7.2.2 [temp.dep.expr]) and it is value-dependent (17.7.2.3 [temp.dep.constexpr]) if and only if *type* is dependent. The result of applying the `offsetof` macro to a field that is a static data member or a function member is undefined.

[Note: the original wording shown here reflects the resolutions of library issues 306 and 449.]

---

197. Issues with two-stage lookup of dependent names

Section: 17.7.4.2 [temp.dep.candidate] **Status:** CD1 **Submitter:** Derek Inglis **Date:** 26 Jan 2000

[Voted into WP at October 2005 meeting.]

The example in 17.7 [temp.res] paragraph 9 is incorrect, according to 17.7.4.2 [temp.dep.candidate] . The example reads,

```
void f(char);

template <class T> void g(T t)
{
    f(1);        // f(char);
    f(T(1));     // dependent
    f(t);        // dependent
    dd++;        // not dependent
                 // error: declaration for dd not found
}

void f(int);

double dd;
void h()
{
    g(2);        // will cause one call of f(char) followed
                 // by two calls of f(int)
    g('a');      // will cause three calls of f(char)
}
```

Since 17.7.4.2 [temp.dep.candidate] says that only Koenig lookup is done from the instantiation context, and since 6.4.2 [basic.lookup.argdep] says that fundamental types have no associated namespaces, either the example is incorrect (and `f(int)` will never be called) or the specification in 17.7.4.2 [temp.dep.candidate] is incorrect.

Notes from 04/00 meeting:

The core working group agreed that the example as written is incorrect and should be reformulated to use a class type instead of a fundamental type. It was also decided to open a new issue dealing more generally with Koenig lookup and fundamental types.

(See also issues 213 and 225.)

Proposed resolution (April, 2005):

Change the example in 17.7 [temp.res] paragraph 9 as follows:

```
void f(char);

template <class T> void g(T t)
{
    f(1);        // f(char);
    f(T(1));     // dependent
    f(t);        // dependent
    dd++;        // not dependent
                 // error: declaration for dd not found
}

enum E { e };
void f(~~int~~E);

double dd;
void h()
{
    g(~~2~~e);       // will cause one call of f(char) followed
                 // by two calls of f(~~int~~E)
    g('a');      // will cause three calls of f(char)
}
```

---

387. Errors in example in 14.6.5

Section: 17.7.5 [temp.inject] **Status:** CD1 **Submitter:** Aleksey Gurtovoy **Date:** 27 Oct 2002

[Voted into WP at March 2004 meeting.]

The example in 17.7.5 [temp.inject] paragraph 2 is incorrect:

```
template<typename T> class number {
    number(int);
    //...
    friend number gcd(number& x, number& y) { /* ... */ }
    //...
};

void g()
{
    number<double> a(3), b(4);
    //...
    a = gcd(a,b);   // finds gcd because number<double> is an
                    // associated class, making gcd visible
                    // in its namespace (global scope)
    b = gcd(3,4);   // ill-formed; gcd is not visible
}
```

Regardless of the last statement ("b = gcd(3,4);"), the above code is ill-formed:

a) number's constructor is private;

b) the definition of (non-void) friend 'gcd' function does not contain a return statement.

Proposed resolution (April 2003):

Replace the example in 17.7.5 [temp.inject] paragraph 2

```
template<typename T> class number {
        number(int);
        //...
        friend number gcd(number& x, number& y) { /* ... */ }
        //...
};

void g()
{
        number<double> a(3), b(4);
        //...
        a = gcd(a,b);           //  finds  gcd  because  number<double>  is an
                                //  associated class, making  gcd  visible
                                //  in its namespace (global scope)
        b = gcd(3,4);           //  ill-formed;  gcd  is not visible
}
```

by

```
template<typename T> class number {
   public:
        number(int);
        //...
        friend number gcd(number x, number y) { return 0; }
   private:
        //...
};

void g()
{
        number<double> a(3), b(4);
        //...
        a = gcd(a,b);           //  finds  gcd  because  number<double>  is an
                                //  associated class, making  gcd  visible
                                //  in its namespace (global scope)
        b = gcd(3,4);           //  ill-formed;  gcd  is not visible
}
```

Drafting note: Added "return" to the friend function, removed references in gcd arguments, added access specifiers.

---

259. Restrictions on explicit specialization and instantiation

Section: 17.8 [temp.spec] **Status:** CD1 **Submitter:** Matt Austern **Date:** 2 Nov 2000

[Moved to DR at 4/02 meeting.]

According to 17.8 [temp.spec] paragraph 5,

> No program shall explicitly instantiate any template more than once, both explicitly instantiate and explicitly specialize a template, or specialize a template more than once for a given set of *template-argument*s.

This rule has an impact on library issue 120. Library authors would like to have the freedom to specialize (or not) various library functions without having to document their choices, while users need the flexibility to explicitly instantiate library functions in certain translation units.

If this rule could be slightly weakened, it would reduce the need for constraining either the library author or the programmer. For instance, the rule might be recast to say that if a specialization is followed by an explicit instantiation in the same translation unit, the explicit instantiation is ignored. A specialization and an explicit instantiation of the same template in two different translation units would still be an error, no diagnostic required.

Proposed resolution (04/01):

1. Replace the first sentence of 17.8 [temp.spec] paragraph 5,

   > No program shall explicitly instantiate any template more than once, both explicitly instantiate and explicitly specialize a template, or specialize a template more than once for a given set of *template-argument*s.

   by

   > For a given template and a given set of *template-argument*s,
   > - an explicit instantiation shall appear at most once in a program,
   > - an explicit specialization shall be defined at most once according to 6.2 [basic.def.odr] in a program, and
   > - both an explicit instantiation and a declaration of an explicit specialization shall not appear in a program unless the explicit instantiation follows a declaration of the explicit specialization.

2. Replace 17.8.2 [temp.explicit] paragraph 4,

   > The definition of a non-exported function template, a non-exported member function template, or a non-exported member function or static data member of a class template shall be present in every translation unit in which it is explicitly instantiated.

   by

   > For a given set of template parameters, if an explicit instantiation of a template appears after a declaration of an explicit specialization for that template, the explicit instantiation has no effect. Otherwise, the definition of a non-exported function template, a non-exported member function template, or a non-exported member function or static data member of a class template shall be present in every translation unit in which it is explicitly instantiated.

---

63. Class instantiation from pointer conversion to void*, null and self

Section: 17.8.1 [temp.inst] **Status:** CD1 **Submitter:** Steve Adamczyk **Date:** 13 Oct 1998

[Moved to DR at October 2002 meeting.]

A template is implicitly instantiated because of a "pointer conversion" on an argument. This was intended to include related-class conversions, but it also inadvertently includes conversions to `void*`, null pointer conversions, cv-qualification conversions and the identity conversion.

It is not clear whether a `reinterpret_cast` of a pointer should cause implicit instantiation.

Proposed resolution (10/01): Replace 17.8.1 [temp.inst] paragraph 4, up to the example, with the following:

> A class template specialization is implicitly instantiated if the class type is used in a context that requires a completely-defined object type or if the completeness of the class type might affect the semantics of the program. [*Note:* in particular, if the semantics of an expression depend on the member or base class lists of a class template specialization, the class template specialization is implicitly generated. For instance, deleting a pointer to class type depends on whether or not the class declares a destructor, and conversion between pointer to class types depends on the inheritance relationship between the two classes involved.]

This version differs from the previous version is its use of the word "might" in the first sentence.

(See also issue 212.)

---

525. Missing `*` in example

Section: 17.8.1 [temp.inst] **Status:** CD1 **Submitter:** Mike Miller **Date:** 25 July 2005

[Voted into WP at April, 2006 meeting.]

The example in 17.8.1 [temp.inst] paragraph 4 has a typographical error: the third parameter of function `g` should be `D<double>* ppp`, but it is missing the `*`:

```
template <class T> class B { /* ... */ };
template <class T> class D : public B<T> { /* ... */ };
```

```
void f(void*);
void f(B<int >*);

void g(D<int>* p, D<char>* pp, D<double> ppp)
{
  f(p);              // instantiation of D<int> required: call f(B<int>*)

  B<char>* q = pp;   // instantiation of D<char> required:
                     // convert D<char>* to B<char>*
  delete ppp;        // instantiation of D<double> required
}
```

Proposed resolution (October, 2005):

As suggested.

---

237. Explicit instantiation and base class members

Section: 17.8.2 [temp.explicit] **Status:** CD1 **Submitter:** Christophe de Dinechin **Date:** 28 Jul 2000

[Voted into WP at October 2005 meeting.]

In 17.8.2 [temp.explicit] paragraph 7 we read:

> The explicit instantiation of a class template specialization implies the instantiation of all of its members not previously explicitly specialized in the translation unit containing the explicit instantiation.

Is "member" intended to mean "non-inherited member?" If yes, maybe it should be clarified since 13 [class.derived] paragraph 1 says,

> Unless redefined in the derived class, members of a base class are also considered to be members of the derived class.

Proposed resolution (October, 2004):

Fixed by the resolution of issue 470.

---

470. Instantiation of members of an explicitly-instantiated class template

Section: 17.8.2 [temp.explicit] **Status:** CD1 **Submitter:** Matt Austern **Date:** 11 May 2004

[Voted into WP at October 2005 meeting.]

17.8.2 [temp.explicit] paragraph 7 says,

> The explicit instantiation of a class template specialization implies the instantiation of all of its members not previously explicitly specialized in the translation unit containing the explicit instantiation.

It's not clear whether this "implied" instantiation is implicit or explicit instantiation. It makes a difference in cases like the following:

```
template <typename T> struct foo {
    struct bar { };
};

template struct foo<int>;         // #1

template struct foo<int>::bar;    // #2
```

If the instantiation of `foo<int>::bar` implied by `#1` is implicit, the explicit instantiation in `#2` is well-formed. Otherwise, `#2` violates the requirement in 17.8 [temp.spec] that

> No program shall explicitly instantiate any template more than once ... for a given set of *template-argument*s.

It's also unclear whether the implied instantiation applies only to direct members of the class template or to inherited members, as well.

John Spicer: I have always interpreted this as meaning only the members declared in the class, not those inherited from other classes. This is what EDG does, and appears to be what g++, Microsoft and Sun do, too. I also think this is the correct thing for the Standard to require. If I were to derive a class from a class in the standard library, an explicit instantiation of my class should not cause the explicit instantiation of things in the standard library (because the library might provide such explicit instantiations, thus causing my program to run afoul of the "can't instantiate more than once" rule).

Proposed resolution (October, 2004):

Change 17.8.2 [temp.explicit] paragraph 7 as follows:

> The explicit instantiation of a class template specialization ~~implies the instantiation of all~~ **also explicitly instantiates each** of its members ~~not~~ **(not including members inherited from base classes) whose definition is visible at the point of instantiation and that has not been** previously explicitly specialized in the translation unit containing the explicit instantiation.

---

551. When is `inline` permitted in an explicit instantiation?

Section: 17.8.2 [temp.explicit] **Status:** CD1 **Submitter:** Steve Clamage **Date:** 07 December 2005

[Voted into the WP at the April, 2007 meeting as part of paper J16/07-0095 = WG21 N2235.]

The Standard does not definitively say when the `inline` specifier may be used in an explicit instantiation. For example, the following would seem to be innocuous, as the function being instantiated is already inline:

```
template <typename T> struct S {
    void f() { }
};
template inline void S<int>::f();
```

However, presumably one would want to prohibit something like:

```
template <typename T> void f(T) { }
template inline void f(int);
```

10.1.2 [dcl.fct.spec] paragraph 4 (after application of the resolution of issue 317) comes close to covering the obvious problematic cases:

> If the definition of a function appears in a translation unit before its first declaration as inline, the program is ill-formed. If a function with external linkage is declared inline in one translation unit, it shall be declared inline in all translation units in which it appears; no diagnostic is required.

This would seem to prohibit the latter case, but apparently would not handle an exported template that was instantiated as inline (because the definition might not appear in the same translation unit as the inline instantiation). It would be better to make a clear statement regarding the use of `inline` in explicit instantiations.

Notes from the April, 2006 meeting:

The CWG favored completely disallowing the `inline` keyword in explicit instantiation directives.

---

44. Member specializations

Section: 17.8.3 [temp.expl.spec] **Status:** CD1 **Submitter:** Nathan Myers **Date:** 19 Sep 1998

[Moved to DR at 4/01 meeting.]

Some compilers reject the following:

```
struct A {
    template <int I> void f();
    template <> void f<0>();
};
```

on the basis of 17.8.3 [temp.expl.spec] paragraph 2:

> An explicit specialization shall be declared in the namespace of which the template is a member, or, for member templates, in the namespace of which the enclosing class or enclosing class template is a member. An explicit specialization of a member function, member class or static data member of a class template shall be declared in the namespace of which the class template is a member. ...

claiming that the specialization above is not "in the namespace of which the enclosing class ... is a member". Elsewhere, declarations are sometimes required to be "at" or "in" "namespace scope", which is not what it says here. Paragraph 17 says:

> A member or a member template may be nested within many enclosing class templates. If the declaration of an explicit specialization for such a member appears in namespace scope, the member declaration shall be preceded by a `template<>` for each enclosing class template that is explicitly specialized.

The qualification "if the declaration ... appears in namespace scope", implies that it might appear elsewhere. The only other place I can think of for a member specialization is in class scope.

Was it the intent of the committee to forbid the construction above? (Note that A itself is not a template.) If so, why?

Proposed resolution (04/01): In-class specializations of member templates are not allowed. In 17.8.3 [temp.expl.spec] paragraph 17, replace

> If the declaration of an explicit specialization for such a member appears in namespace scope...

with

> In an explicit specialization for such a member...

Notes from 04/00 meeting:

This issue was kept in "review" status for two major reasons:

1. It's not clear that a change is actually needed. All uses of the phrase "in the namespace" in the IS mean "directly in the namespace," not in a scope nested within the namespace.
2. There was substantial sentiment for actually adding support for in-class specializations at a future time, and it might be perceived as a reversal of direction to pass a change aimed at reinforcing the absence of the feature, only to turn around afterward and add it.

Notes from 10/00 meeting:

The core working group felt that the value of additional clarity here outweighs the potential disadvantages that were noted at the preceding meeting.

---

275. Explicit instantiation/specialization and *using-directive*s

Section: 17.8.3 [temp.expl.spec] **Status:** CD1 **Submitter:** John Spicer **Date:** 15 Feb 2001

[Moved to DR at 4/02 meeting.]

Consider this example:

```
namespace N {
    template <class T> void f(T){}
    template <class T> void g(T){}
    template <> void f(int);
    template <> void f(char);
    template <> void g(char);
}

using namespace N;

namespace M {
    template <> void N::f(char){}  // prohibited by standard
    template <class T> void g(T){}
    template <> void g(char){}     // specialization of M::g or ambiguous?
    template void f(long);         // instantiation of N::f?
}

template <class T> void g(T){}

template <> void N::f(char){}  // okay
template <> void f(int){}      // is this a valid specialization of N::f?

template void g(int);          // instantiation of ::g(int) or ambiguous?
```

The question here is whether unqualified names made visible by a *using-directive* can be used as the declarator in an explicit instantiation or explicit specialization.

Note that this question is already answered for qualified names in 11.3 [dcl.meaning] paragraph 1. In a qualified name such as `N::f`, `f` must be a member of class or namespace `N`, not a name made visible in `N` by a *using-directive* (or a *using-declaration*, for that matter).

The standard does not, as far as I can tell, specify the behavior of these cases one way or another.

My opinion is that names from *using-directive*s should not be considered when looking up the name in an unqualified declarator in an explicit specialization or explicit instantiation. In such cases, it is reasonable to insist that the programmer know exactly which template is being specialized or instantiated, and that a qualified name must be used if the template is a member of a namespace.

As the example illustrates, allowing names from *using-directive*s to be used would also have the affect of making ambiguous otherwise valid instantiation and specialization directives.

Furthermore, permitting names from *using-directive*s would require an additional rule to prohibit the explicit instantiation of an entity in one namespace from being done in another (non-enclosing) namespace (as in the instantiation of `f` in namespace `M` in the example).

Mike Miller: I believe the explicit specialization case is already covered by 10.3.1.2 [namespace.memdef] paragraph 2, which requires using a qualified name to define a namespace member outside its namespace.

John Spicer: 10.3.1.2 [namespace.memdef] deals with namespace members. An explicit specialization directive deals with something that is a specialization of a namespace member. I don't think the rules in 10.3.1.2 [namespace.memdef] could be taken to apply to specializations unless the standard said so explicitly.

Proposed resolution (suggested 04/01, proposed 10/01):

(The first change below will need to be revised in accordance with the resolution of issue 284 to add a cross-reference to the text dealing with class names.)

1. Add in 17.8.2 [temp.explicit] paragraph 2 before the example:

   > An explicit instantiation shall appear in an enclosing namespace of its template. If the name declared in the explicit instantiation is an unqualified name, the explicit instantiation shall appear in the namespace where its template is declared. [*Note:* Regarding qualified names in declarators, see 11.3 [dcl.meaning].]

2. Change the first sentence of 10.3.1.2 [namespace.memdef] paragraph 1 from

Members of a namespace can be defined within that namespace.

to

Members (including explicit specializations of templates (17.8.3 [temp.expl.spec])) of a namespace can be defined within that namespace.

3. Change the first sentence of 10.3.1.2 [namespace.memdef] paragraph 2 from

Members of a named namespace can also be defined...

to

Members (including explicit specializations of templates (17.8.3 [temp.expl.spec])) of a named namespace can also be defined...

4. Change the last sentence of 17.8.3 [temp.expl.spec] paragraph 2 from

If the declaration is not a definition, the specialization may be defined later in the namespace in which the explicit specialization was declared, or in a namespace that encloses the one in which the explicit specialization was declared.

to

If the declaration is not a definition, the specialization may be defined later (10.3.1.2 [namespace.memdef]).

---

336. Explicit specialization examples are still incorrect

Section: 17.8.3 [temp.expl.spec] **Status:** CD1 **Submitter:** Jason Shirk **Date:** 29 Jan 2002

[Voted into WP at April 2003 meeting.]

The examples corrected by issue 24 are still wrong in one case.

In item #4 (a correction to the example in paragraph 18), the proposed resolution is:

```
template<class T1> class A {
  template<class T2> class B {
    template<class T3> void mf1(T3);
      void mf2();
    };
};
template<> template<class X>
  class A<int>::B { };
template<> template<> template<class T>
  void A<int>::B<double>::mf1(T t) { }
template<class Y> template<>
  void A<Y>::B<double>::mf2() { } // ill-formed; B<double> is specialized but
                                  // its enclosing class template A is not
```

The explicit specialization of member `A<int>::B<double>::mf1` is ill-formed. The class template `A<int>::B` is explicitly specialized and contains no members, so any implicit specialization (such as `A<int>::B<double>`) would also contain no members.

Proposed Resolution (4/02):

Fix the example in 17.8.3 [temp.expl.spec] paragraph 18 to read:

```
template<class T1> class A {
  template<class T2> class B {
    template<class T3> void mf1(T3);
    void mf2();
  };
};
template<> template<class X>
  class A<int>::B {
    template<class T> void mf1(T);
  };
template<> template<> template<class T>
  void A<int>::B<double>::mf1(T t) { }
template<class Y> template<>
  void A<Y>::B<double>::mf2() { } // ill-formed; B<double> is specialized but
                                  // its enclosing class template A is not
```

---

337. Attempt to create array of abtract type should cause deduction to fail

Section: 17.9.2 [temp.deduct] **Status:** CD1 **Submitter:** John Spicer **Date:** 30 Jan 2002

[Voted into WP at April 2003 meeting.]

In 17.9.2 [temp.deduct], attempting to create an array of abstract class type should be included in the list of things that cause type deduction to fail.

Proposed Resolution (4/02):

In 17.9.2 [temp.deduct] paragraph 2 amend the bullet item:

> Attempting to create an array with an element type that is `void`, a function type, or a reference type, or attempting to create an array with a size that is zero or negative.

To the following:

> Attempting to create an array with an element type that is `void`, a function type, ~~or~~ a reference type, **or an abstract class type,** or attempting to create an array with a size that is zero or negative.

---

368. Uses of non-type parameters that should cause deduction to fail

Section: 17.9.2 [temp.deduct] **Status:** CD1 **Submitter:** Jason Shirk **Date:** 29 July 2002

[Voted into WP at October 2003 meeting.]

I understand the rules in 17.9.2 [temp.deduct] paragraph 2 are meant to be an exhaustive list of what can cause type deduction to fail.

Consider:

```
template<typename U,U u> struct wrap_t;

template<typename U> static yes check( wrap_t<U,U(0)>* );

struct X { X(int); };
int main() {
  check<X>(0);
}
```

I can see 2 reasons this might cause type deduction to fail:

- Conversion is not a compile time constant.
- Non-type parameter is a class type.

Neither case is mentioned in 17.9.2 [temp.deduct] paragraph 2, nor do I see a DR mentioning these.

Proposed resolution (October 2002):

Add after the fourth-to-last bullet of 17.9.2 [temp.deduct] paragraph 2:

- Attempting to give an invalid type to a nontype template parameter. [*Example:*

  ```
  template <class T, T> struct S {};
  template <class T> int f(S<T, T()>*);
  struct X {};
  int i0 = f<X>(0);
  ```

  ]

---

398. Ambiguous wording on naming a type in deduction

Section: 17.9.2 [temp.deduct] **Status:** CD1 **Submitter:** Daveed Vandevoorde **Date:** 16 Jan 2003

[Voted into WP at March 2004 meeting.]

The following example (simplified from a posting to comp.lang.c++.moderated) is accepted by some compilers (e.g., EDG), but not by other (e.g., g++).

```
struct S {
  static int const I = 42;
};

template<int N> struct X {};

template<typename T> void f(X<T::I>*) {}

template<typename T> void f(X<T::J>*) {}

int main() {
  f<S>(0);
}
```

The wording in the standard that normally would cover this (third sub-bullet in 17.9.2 [temp.deduct] paragraph 2) says:

> Attempting to use a type in the qualifier portion of a qualified name **that names a type** when that type does not contain the specified member, or if the specified member is not a type where a type is required.

(emphasis mine). If the phrase "that names a type" applies to "a qualified name," then the example is invalid. If it applies to "the qualifier portion," then it is valid (because the second candidate is simply discarded).

I suspect we want this example to work. Either way, I believe the sub-bullet deserves clarification.

Notes from April 2003 meeting:

We agreed that the example should be valid. The phrase "that names a type" applies to "the qualifier portion."

Proposed resolution (October 2003):

In 17.9.2 [temp.deduct], paragraph 2, bullet 3, sub-bullet 3, replace

> Attempting to use a type in the qualifier portion of a qualified name that names a type when that type does not contain the specified member, or if the specified member is not a type where a type is required.

With

> Attempting to use a type in a *nested-name-specifier* of a *qualified-id* when that type does not contain the specified member, or
>
> - the specified member is not a type where a type is required, or
> - the specified member is not a template where a template is required, or
> - the specified member is not a nontype where a nontype is required.
>
> [*Example:*

Replace the example that follows the above text with

```
template <int I> struct X { };
template <template <class T> class> struct Z {};
template <class T> void f(typename T::Y*){}
template <class T> void g(X<T::N>*){}
template <class T> void h(Z<T::template TT>*){}
struct A {};
struct B { int Y; };
struct C {
        typedef int N;
};
struct D {
        typedef int TT;
};

int main()
{
        // Deduction fails in each of these cases:
        f<A>(0); // A does not contain a member Y
        f<B>(0); // The Y member of B is not a type
        g<C>(0); // The N member of C is not a nontype
        h<D>(0); // The TT member of D is not a template
}
```

>]

---

486. Invalid return types and template argument deduction

Section: 17.9.2 [temp.deduct] **Status:** CD1 **Submitter:** John Spicer **Date:** 16 Nov 2004

[Voted into WP at April, 2006 meeting.]

According to 17.9.2 [temp.deduct] paragraph 2,

> If a substitution in a template parameter or in the function type of the function template results in an invalid type, type deduction fails.

That would seem to apply to cases like the following:

```
template <class T> T f(T&){}
void f(const int*){}
int main() {
  int a[5];
  f(a);
}
```

Here, the return type of `f` is deduced as `int[5]`, which is invalid according to 11.3.5 [dcl.fct] paragraph 6. The outcome of this example, then, should presumably be that type deduction fails and overload resolution selects the non-template function. However, the list of reasons in 17.9.2 [temp.deduct] for which type deduction can fail does not include function and array types as a function return type. Those cases should be added to the list.

Proposed resolution (October, 2005):

Change the last sub-bullet of 17.9.2 [temp.deduct] paragraph 2 as indicated:

- Attempting to create a function type in which a parameter has a type of `void`**, or in which the return type is a function type or array type**.

488. Local types, overload resolution, and template argument deduction

Section: 17.9.2 [temp.deduct] **Status:** CD1 **Submitter:** Mark Mitchell **Date:** 24 Nov 2004

[Voted into the WP at the June, 2008 meeting as paper N2657.]

It is not clear how to handle the following example:

```
struct S {
    template <typename T> S(const T&);
};
void f(const S&);
void f(int);
void g() {
    enum E { e };
    f(e);    // ill-formed?
}
```

Three possibilities suggest themselves:

1. **Fail during overload resolution.** In order to perform overload resolution for the call to `f`, the declaration of the required specialization of the `S` constructor must be instantiated. This instantiation uses a local type and is thus ill-formed (17.3.1 [temp.arg.type] paragraph 2), rendering the example as a whole ill-formed, as well.

2. **Treat this as a type-deduction failure.** Although it is not listed currently among the causes of type-deduction failure in 17.9.2 [temp.deduct] paragraph 2, it could plausibly be argued that instantiating a function declaration with a local type as a template type-parameter falls under the rubric of "If a substitution in a template parameter or in the function type of the function template results in an invalid type" and thus should be a type-deduction failure. The result would be that the example is well-formed because `f(const S&)` would be removed from the list of viable functions.

3. **Fail only if the function selected by overload resolution requires instantiation with a local type.** This approach would require that the diagnostic resulting from the instantiation of the function type during overload resolution be suppressed and either regenerated or regurgitated once overload resolution is complete. (The example would be well-formed under this approach because `f(int)` would be selected as the best match.)

(See also issue 489.)

Notes from the April, 2005 meeting:

The question in the original example was whether there should be an error, even though the uninstantiable template was not needed for calling the best-matching function. The broader issue is whether a user would prefer to get an error or to call a "worse" non-template function in such cases. For example:

```
template<typename T> void f(T);
void f(int);
void g() {
    enum E { e };
    f(e);    // call f(int) or get an error?
}
```

It was observed that the type deduction rules are intended to model, albeit selectively, the other rules of the language. This would argue in favor of the second approach, a type-deduction failure, and the consensus of the group was that the incremental benefit of other approaches was not enough to outweigh the additional complexity of specification and implementation.

Proposed resolution (October, 2005):

Add a new sub-bullet following bullet 3, sub-bullet 7 ("Attempting to give an invalid type to a non-type template parameter") of 17.9.2 [temp.deduct] paragraph 2:

- Attempting to use a local or unnamed type as the value of a template type parameter.

Additional note (December, 2005):

The Evolution Working Group is currently considering an extension that would effectively give linkage to some (but perhaps not all) types that currently have no linkage. If the proposed resolution above is adopted and then later a change along the lines that the EWG is considering were also adopted, the result would be a silent change in the result of overload resolution, because the newly-acceptable specializations would become part of the overload set. It is not clear whether that possibility is sufficient reason to delay adoption of this resolution or not.

Notes from the April, 2007 meeting:

The Evolution Working Group is now actively pursuing an extension that would allow local and/or nameless types to be used as template arguments, so this resolution will be held in abeyance until the outcome of that proposal is known.

Notes from the June, 2008 meeting:

Paper N2657, adopted at the Sophia Antipolis (June, 2008) meeting, removed the restriction against local and unnamed types as template parameters. The example is now well-formed.

352. Nondeduced contexts

Section: 17.9.2.1 [temp.deduct.call] **Status:** CD1 **Submitter:** Andrei Iltchenko **Date:** 24 April 2002

[Voted into WP at March 2004 meeting.]

The current definition of the C++ language speaks about nondeduced contexts only in terms of deducing template arguments from a type 17.9.2.5 [temp.deduct.type] paragraph 4. Those cases, however, don't seem to be the only ones when template argument deduction is not possible. The example below illustrates that:

```
namespace  A  {
   enum  ae  {  };
   template<class R, class A>
   int  foo(ae, R(*)(A))
   {  return  1;  }
}

template<typename T>
void  tfoo(T)
{  }

template<typename T>
int  tfoo(T)
{  return  1;  }

/*int  tfoo(int)
{  return  1;  }*/


int  main()
{
   A::ae  a;
   foo(a, &tfoo);
}
```

Here argument-dependent name lookup finds the function template 'A::foo' as a candidate function. None of the function template's function parameter types constitutes a nondeduced context as per 17.9.2.5 [temp.deduct.type] paragraph 4. And yet, quite clearly, argument deduction is not possible in this context. Furthermore it is not clear what a conforming implementation shall do when the definition of the non-template function '::tfoo' is uncommented.

Suggested resolution:

Add the following as a new paragraph immediately before paragraph 3 of 17.9.2.1 [temp.deduct.call]:

> After the above transformations, in the event of P being a function type, a pointer to function type, or a pointer to member function type and the corresponding A designating a set of overloaded (member) functions with at least one of the (member) functions introduced by the use of a (member) function template name (16.4 [over.over]) or by the use of a conversion function template (17.9.2.3 [temp.deduct.conv]), the whole function call expression is considered to be a nondeduced context. [Example:
>
> ```
> namespace  A  {
>    enum  ae  {  };
>    template<class R, class A>
>    int  foo(ae, R(*)(A))
>    {  return  1;  }
> }
>
> template<typename T>
> void  tfoo(T)
> {  }
>
> template<typename T>
> int  tfoo(T)
> {  return  1;  }
>
> int  tfoo(int)
> {  return  1;  }
>
> int  main()
> {
>    A::ae  a;
>    foo(a, &tfoo);  //  ill-formed, the call is a nondeduced context
>    using  A::foo;
>    foo<void,int>(a, &tfoo);  // well-formed, the address of the spe-
>                              // cialization 'void tfoo<int>(int)' is
>                              // the second argument of the call
> }
> ```

Notes from October 2002 meeting:

There was agreement that deduction should fail but it's still possible to get a result -- it's just not a "nondeduced context" in the sense of the standard.

The presence of a template in the overload set should not automatically disqualify the overload set.

Proposed Resolution (April 2003, revised October 2003):

In 17.9.2.5 [temp.deduct.type] paragraph 4 replace:

> The nondeduced contexts are:

- The *nested-name-specifier* of a type that was specified using a *qualified-id*.
- A type that is a *template-id* in which one or more of the *template-arguments* is an expression that references a *template-parameter*.

with:

The nondeduced contexts are:

- The *nested-name-specifier* of a type that was specified using a *qualified-id*.
- A non-type template argument or an array bound that is an expression that references a template-parameter.
- A template parameter used in the parameter type of a function parameter that has a default argument that is being used in the call for which argument deduction is being done.
- A function parameter for which argument deduction cannot be done because the associated function argument is a function, or a set of overloaded functions (16.4 [over.over]), and one or more of the following apply:
  - more than one function matches the function parameter type (resulting in an ambiguous deduction), or
  - no function matches the function parameter type, or
  - the set of functions supplied as an argument contains one or more function templates.

In 17.9.2.1 [temp.deduct.call], add after paragraph 3:

When P is a function type, pointer to function type, or pointer to member function type:

- If the argument is an overload set containing one or more function templates, the parameter is treated as a nondeduced context.
- If the argument is an overload set (not containing function templates), trial argument deduction is attempted using each of the members of the set. If deduction succeeds for only one of the overload set members, that member is used as the argument value for deduction. If deduction succeeds for more than one member of the overload set the parameter is treated as a nondeduced context.

[Example:

```
// Only one function of an overload set matches the call so the function
// parameter is a deduced context.
template <class T> int f(T (*p)(T));
int g(int);
int g(char);
int i = f(g);  // calls f(int (*)(int))
```

--end example]

[Example:

```
// Ambiguous deduction causes the second function parameter to be a
// nondeduced context.
template <class T> int f(T, T (*p)(T));
int g(int);
char g(char);
int i = f(1, g);  // calls f(int, int (*)(int))
```

--end example]

[Example:

```
// The overload set contains a template, causing the second function
// parameter to be a nondeduced context.
template <class T> int f(T, T (*p)(T));
char g(char);
template <class T> T g(T);
int i = f(1, g);  // calls f(int, int (*)(int))
```

--end example]

In 17.9.2.5 [temp.deduct.type] paragraph 14, replace:

If, in the declaration of a function template with a non-type *template-parameter*, the non-type *template-parameter* is used in an expression in the function parameter-list, the corresponding *template-argument* must always be explicitly specified or deduced elsewhere because type deduction would otherwise always fail for such a *template-argument*.

With:

If, in the declaration of a function template with a non-type template parameter, the non-type template parameter is used in an expression in the function parameter list, the expression is a nondeduced context.

Replace the example with:

[Example:

```
template<int i> class A { /* ... */ };
template<int i> void g(A<i+1>);
template<int i> void f(A<i>, A<i+1>);
void k() {
  A<1> a1;
  A<2> a2;
  g(a1);  //error: deduction fails for expression i+1
  g<0>(a1); //OK
  f(a1, a2);  // OK
}
```

--end example]

In 17.9.2.5 [temp.deduct.type] paragraph 16, replace:

> A *template-argument* can be deduced from a pointer to function or pointer to member function argument if the set of overloaded functions does not contain function templates and at most one of a set of overloaded functions provides a unique match.

With:

> A *template-argument* can be deduced from a function, pointer to function, or pointer to member function type.

---

522. Array-to-pointer decay in template argument deduction

Section: 17.9.2.1 [temp.deduct.call] **Status:** CD1 **Submitter:** Daveed Vandevoorde **Date:** 3 June 2005

[Voted into WP at the October, 2006 meeting.]

Consider the following example:

```
char* cmdline3_[1] = {};

template<class charT>
void func(const charT* const argv[]) {}

int main()
{
    func(cmdline3_);
}
```

In terms of the process described in 17.9.2.1 [temp.deduct.call], `P` is `const charT* const *` and `A` is `char*[1]`. According to the first bullet in paragraph 2, the type used in deduction is not `A` but "the pointer type produced by the array-to-pointer standard conversion."

According to paragraph 4,

> In general, the deduction process attempts to find template argument values that will make the deduced `A` identical to `A` (after the type `A` is transformed as described above). However, there are three cases that allow a difference:

In this example, the deduced `A` is *not* identical to the transformed `A`, because the deduced `A` has additional cv-qualification, so the three exceptions must be examined to see if they apply. The only one that might apply is the second bullet of paragraph 4:

- `A` can be another pointer or pointer to member type that can be converted to the deduced `A` via a qualification conversion (7.5 [conv.qual]).

However, `A` is not a pointer type but an array type; this provision does not apply and deduction fails.

It has been argued that the phrase "after the type `A` is transformed as described above" should be understood to apply to the `A` in the three bullets of paragraph 4. If that is the intent, the wording should be changed to make that explicit.

Proposed resolution (October, 2005):

Add the indicated words to 17.9.2.1 [temp.deduct.call] paragraph 4:

> In general, the deduction process attempts to find template argument values that will make the deduced `A` identical to `A` (after the type `A` is transformed as described above). However, there are three cases that allow a difference:
>
> - If the original `P` is a reference type, the deduced `A` (i.e., the type referred to by the reference) can be more cv-qualified than **the transformed** `A`.
> - **The transformed** `A` can be another pointer or pointer to member type that can be converted to the deduced `A` via a qualification conversion (7.5 [conv.qual]).
> - If `P` is a class, and `P` has the form *template-id*, then **the transformed** `A` can be a derived class of the deduced `A`. Likewise, if `P` is a pointer to a class of the form *template-id*, **the transformed** `A` can be a pointer to a derived class pointed to by the deduced `A`.

---

606. Template argument deduction for rvalue references

Section: 17.9.2.1 [temp.deduct.call] **Status:** CD1 **Submitter:** Peter Dimov **Date:** 1 December 2006

[Voted into the WP at the September, 2008 meeting as part of paper N2757.]

There are a couple of minor problems with the rvalue reference wording in the WP. The non-normative note in 17.9.2.1 [temp.deduct.call] paragraph 3 says,

[*Note:* The effect of this rule for lvalue arguments and rvalue reference parameters is that deduction in such cases will fail unless the function parameter is of the form *cv* `T&&` (17.9.2.5 [temp.deduct.type]). —*end note*]

It turns out that this isn't correct. For example:

```
template <class T> void g(basic_string<T> && );
...
basic_string<char> s;
g(s);     // Note says that it should fail, we want it to call
          // g<char>(basic_string<char>&&)
```

Additionally, consider this case:

```
template <class T> void f(const T&&);
...
int i;
f(i);
```

If we deduce `T` as `int&` in this case then `f(i)` calls `f<int&>(int&)`, which seems counterintuitive. We prefer that `f<int>(const int&&)` be called. Therefore, we would like the wording clarified that the `A&` deduction rule in 17.9.2.1 [temp.deduct.call] paragraph 3 applies only to the form `T&&` and not to *cv* `T&&` as the note currently implies.

These are minor tweaks to the rvalue reference wording and a fallout from issue 540. In particular, the major applications of move semantics and perfect forwarding are not impacted with respect to the original intentions of the rvalue reference work by these suggestions.

Suggested resolution:

Change 17.9.2.1 [temp.deduct.call] paragraph 3 as follows:

If `P` is ~~an rvalue reference type~~ **of the form** `T&&`**, where** `T` **is a template parameter,** and the argument is an lvalue, ~~the type `A&` is used in place of `A` for type deduction~~ `T` **is deduced as** `A&`. [*Example:*

```
template <typename T> int f(T&&);
int i;
int j = f(i); // calls f<int&>(i)
template <typename T> int g(const T&&);
 int k;
 int n = g(k); // calls g<int>(k)
```

—*end example*][~~*Note:* The effect of this rule for lvalue arguments and rvalue reference parameters is that deduction in such cases will fail unless the function parameter is of the form *cv* `T&&` (17.9.2.5 [temp.deduct.type]). —*end note*~~]

Proposed resolution (August, 2008):

Change 17.9.2.1 [temp.deduct.call] paragraph 3 as follows:

If `P` is ~~an rvalue reference type~~ **of the form** `T&&`**, where** `T` **is a template parameter,** and the argument is an lvalue, the type `A&` is used in place of `A` for type deduction. [*Example:*

```
template <typename T> int f(T&&);
int i;
int j = f(i); // calls f<int&>(i)
template <typename T> int g(const T&&);
 int k;
 int n = g(k); // calls g<int>(k)
```

—*end example*][~~*Note:* The effect of this rule for lvalue arguments and rvalue reference parameters is that deduction in such cases will fail unless the function parameter is of the form *cv* `T&&` (17.9.2.5 [temp.deduct.type]). —*end note*~~]

---

322. Deduction of reference conversions

Section: 17.9.2.3 [temp.deduct.conv] **Status:** CD1 **Submitter:** Jason Merrill **Date:** 14 Nov 2001

[Voted into WP at April 2003 meeting.]

Consider:

```
struct S {
  template <class T> operator T& ();
};

int main ()
{
  S s;
  int i = static_cast<int&> (s);
}
```

17.9.2.3 [temp.deduct.conv] says that we strip the reference from `int&`, but doesn't say anything about `T&`. As a result, `P` (`T&`) and `A` (`int`) have incompatible forms and deduction fails.

Proposed Resolution (4/02):

Change the last chunk of 17.9.2.3 [temp.deduct.conv] paragraph 2 from

If A is a cv-qualified type, the top level cv-qualifiers of A's type are ignored for type deduction. If A is a reference type, the type referred to by A is used for type deduction.

to

If A is a cv-qualified type, the top level cv-qualifiers of A's type are ignored for type deduction. If A is a reference type, the type referred to by A is used for type deduction. If P is a reference type, the type referred to by P is used for type deduction.

---

349. Template argument deduction for conversion functions and qualification conversions

Section: 17.9.2.3 [temp.deduct.conv] **Status:** CD1 **Submitter:** John Spicer **Date:** 16 April 2002

[Voted into WP at October 2003 meeting.]

We ran into an issue concerning qualification conversions when doing template argument deduction for conversion functions.

The question is: What is the type of T in the conversion functions called by this example? Is T "int" or "const int"?

If T is "int", the conversion function in class A works and the one in class B fails (because the return expression cannot be converted to the return type of the function). If T is "const int", A fails and B works.

Because the qualification conversion is performed on the result of the conversion function, I see no benefit in deducing T as const int.

In addition, I think the code in class A is more likely to occur than the code in class B. If the author of the class was planning on returning a pointer to a const entity, I would expect the function to have been written with a const in the return type.

Consequently, I believe the correct result should be that T is int.

```
struct A {
        template <class T> operator T***() {
                int*** p = 0;
                return p;
        }
};

struct B {
        template <class T> operator T***() {
                const int*** p = 0;
                return p;
        }
};

int main()
{
        A a;
        const int * const * const * p1 = a;
        B b;
        const int * const * const * p2 = b;
}
```

We have just implemented this feature, and pending clarification by the committee, we deduce T as int. It appears that g++ and the Sun compiler deduce T as const int.

One way or the other, I think the standard should be clarified to specify how cases like this should be handled.

Notes from October 2002 meeting:

There was consensus on having the deduced type be "int" in the above.

Proposed resolution (April 2003):

Add to the end of 17.9.2.3 [temp.deduct.conv] (as a new paragraph following paragraph 3):

When the deduction process requires a qualification conversion for a pointer or pointer to member type as described above, the following process is used to determine the deduced template argument values:

If A is a type $cv_{1,0}$ pointer to ... $cv_{1,n-1}$ pointer to $cv_{1,n}$ *T1*

and P is a type $cv_{2,0}$ pointer to ... $cv_{2,n-1}$ pointer to $cv_{2,n}$ *T2*

The cv-unqualified T1 and T2 are used as the types of A and P respectively for type deduction.

[*Example:*

```
struct A {
        template <class T> operator T***();
};

A a;
const int * const * const * p1 = a;  // T is deduced as int, not const int
```

-- *end example*]

---

70. Is an array bound a nondeduced context?

Section: 17.9.2.5 [temp.deduct.type] **Status:** CD1 **Submitter:** Jack Rouse **Date:** 29 Sep 1998

[Moved to DR at 4/01 meeting.]

Paragraph 4 lists contexts in which template formals are not deduced. Were template formals in an expression in the array bound of an array type specification intentionally left out of this list? Or was the intent that such formals always be explicitly specified? Otherwise I believe the following should be valid:

```
template <int I> class IntArr {};

template <int I, int J>
void concat( int (&d)[I+J], const IntArr<I>& a, const IntArr<J>& b ) {}

int testing()
{
    IntArr<2> a;
    IntArr<3> b;
    int d[5];

    concat( d, a, b );
}
```

Can anybody shed some light on this?

From John Spicer:

Expressions involving nontype template parameters are nondeduced contexts, even though they are omitted from the list in 17.9.2.5 [temp.deduct.type] paragraph 4. See 17.9.2.5 [temp.deduct.type] paragraphs 12-14:

12. A template type argument cannot be deduced from the type of a non-type *template-argument*.

    ...

14. If, in the declaration of a function template with a non-type *template-parameter*, the non-type *template-parameter* is used in an expression in the function parameter-list, the corresponding *template-argument* must always be explicitly specified or deduced elsewhere because type deduction would otherwise always fail for such a *template-argument*.

Proposed resolution (04/01): In 17.9.2.5 [temp.deduct.type] paragraph 4, add a third bullet:

- An array bound that is an expression that references a *template-parameter*

---

300. References to functions in template argument deduction

Section: 17.9.2.5 [temp.deduct.type] **Status:** CD1 **Submitter:** Andrei Iltchenko **Date:** 11 Jul 2001

[Moved to DR at October 2002 meeting.]

Paragraph 9 of 17.9.2.5 [temp.deduct.type] enumerates the forms that the types P and A need to have in order for template argument deduction to succeed.

For P denoting a pointer to function the paragraph lists the following forms as allowing for template argument deduction:

```
type(*)(T)
T(*)()
T(*)(T)
```

On the other hand, no provision has been made to accommodate similar cases for references to functions, which in light of the wording of 17.9.2.5 [temp.deduct.type] paragraph 11 means that the program below is ill-formed (some of the C++ compilers do not reject it however):

```
template<typename Arg, typename Result, typename T>
Result  foo_r(Result(& rf)(Arg), T x)
{   return  rf(Arg(x));   }

template<typename Arg, typename Result, typename T>
Result  foo_p(Result(* pf)(Arg), T x)
{   return  pf(Arg(x));   }

#include <iostream>
int  show_arg(char c)
{
   std::cout << c << ' ';
   if(std::cout)  return  0;
   return  -1;
}

int  main()
{
                                          // The deduction
   int  (& rf1)(int(&)(char), double) = foo_r; // shall fail here
```

```
                                        // While here
    int  (& rf2)(int(*)(char), double) = foo_p; // it shall succeed
    return  rf2(show_arg, 2);
}
```

Proposed resolution (10/01, same as suggested resolution):

In the list of allowable forms for the types `P` and `A` in paragraph 9 of 17.9.2.5 [temp.deduct.type] replace

```
type(*)(T)
T(*)()
T(*)(T)
```

by

```
type(T)
T()
T(T)
```

---

526. Confusing aspects in the specification of non-deduced contexts

Section: 17.9.2.5 [temp.deduct.type] **Status:** CD1 **Submitter:** Mike Miller **Date:** 25 July 2005

[Voted into WP at the October, 2006 meeting.]

17.9.2.5 [temp.deduct.type] paragraph 5 reads:

> The non-deduced contexts are:
>
> - The *nested-name-specifier* of a type that was specified using a *qualified-id*.
> - A non-type template argument or an array bound that is an expression that references a template parameter.
> - A template parameter used in the parameter type of a function parameter that has a default argument that is being used in the call for which argument deduction is being done.
> - A function parameter for which argument deduction cannot be done because the associated function argument is a function, or a set of overloaded functions (16.4 [over.over]), and one or more of the following apply:
>   - more than one function matches the function parameter type (resulting in an ambiguous deduction), or
>   - no function matches the function parameter type, or
>   - the set of functions supplied as an argument contains one or more function templates.
> - An array bound that is an expression that references a *template-parameter*.

There are two problems with this list:

1. The last bullet is redundant with the second bullet. This appears to have been the result of applying the resolutions of issues 70 and 352 independently instead of in coordination.
2. The second bullet appears to be contradicted by the statement in paragraph 8 saying that an argument can be deduced if `P` and `A` have the forms *type*`[i]` and *template-name*`<i>`.

   The intent of the wording in bullet 2 appears to have been that deduction cannot be done if the template parameter is a sub-expression of the template argument or array bound expression and that it can be done if it is the complete expression, but the current wording does not say that very clearly. (Similar wording also appears in 17.7.2.1 [temp.dep.type] paragraph 3 and 17.9.2.5 [temp.deduct.type] paragraph 14.)

Proposed resolution (October, 2005):

1. Change 17.9.2.5 [temp.deduct.type] paragraph 5 as indicated:

   > The non-deduced contexts are:
   >
   > - The *nested-name-specifier* of a type that was specified using a *qualified-id*.
   > - A non-type template argument or an array bound ~~that is an expression that~~ **in either of which a subexpression** references a template parameter.
   > - A template parameter used in the parameter type of a function parameter that has a default argument that is being used in the call for which argument deduction is being done.
   > - A function parameter for which argument deduction cannot be done because the associated function argument is a function, or a set of overloaded functions (16.4 [over.over]), and one or more of the following apply:
   >   - more than one function matches the function parameter type (resulting in an ambiguous deduction), or
   >   - no function matches the function parameter type, or

- the set of functions supplied as an argument contains one or more function templates.

- ~~An array bound that is an expression that references a *template-parameter*.~~

2. Change 17.9.2.5 [temp.deduct.type] paragraph 14 as indicated:

   If, in the declaration of a function template with a non-type template parameter, the non-type template parameter is used in ~~an expression~~ **a subexpression** in the function parameter list, the expression is a non-deduced context **as specified above**...

3. Change 17.7.2.1 [temp.dep.type] paragraph 3 as indicated:

   A template argument that is equivalent to a template parameter (i.e., has the same constant value or the same type as the template parameter) can be used in place of that template parameter in a reference to the current instantiation. In the case of a non-type template argument, the argument must have been given the value of the template parameter and not an expression ~~involving~~ **that contains** the template parameter **as a subexpression**...

---

415. Template deduction does not cause instantiation

Section: 17.9.3 [temp.over] **Status:** CD1 **Submitter:** John Spicer **Date:** 4 May 2003

[Voted into WP at the October, 2006 meeting.]

Mike Miller: In fact, now that I've looked more closely, that appears not to be the case. (At least, it's not the error I get when I compile his example.) Here's a minimal extract (without the inflammatory using-directive :-) that illustrates what I think is going on:

```
template <typename _Iterator>
struct iterator_traits {
  typedef typename _Iterator::difference_type difference_type;
};

template <typename _InputIterator>
inline typename iterator_traits<_InputIterator>::difference_type
distance(_InputIterator, _InputIterator);

double distance(const int&, const int&);

void f() {
  int i = 0;
  int j = 0;
  double d = distance(i, j);
}
```

What happens is that `iterator_traits<int>` is instantiated as part of type deduction for the function template `distance`, and the instantiation fails. (Note that it can't be instantiation of `distance<int>`, as I had originally posited, because in this case only a declaration, not a definition, of that template is in scope.)

John Spicer: Yes, I believe that is what is going on.

Mike Miller: I seem to recall that there was some discussion of questions related to this during the core meetings in Oxford. I think Steve Adamczyk said something to the effect that it's infeasible to suppress all instantiation errors during template type deduction and simply call any such errors a deduction failure. (I could be misremembering, and I could be misapplying that comment to this situation.)

John Spicer: Regardless of other conditions in which this may apply, I don't think it would be reasonable for compilers to have to do "speculative instantiations" during template argument deduction. One class instantiation could kick off a series of other instantiations, etc.

Mike Miller: I don't see anything in the Standard that tells me whether it's legitimate or not to report an error in this case. I hope John or another template expert can enlighten me on that.

John Spicer: My opinion is that, because this case is not among those enumerated that cause deduction failure (rather than being ill-formed) that reporting an error is the right thing to do.

Mike Miller: I am still interested, though, in the question of why 17.9.3 [temp.over] says that viable function template specializations are instantiated, even if they are not selected by overload resolution.

John Spicer: I believe the wording in 17.9.3 [temp.over] is incorrect. I researched this and found that a change was made during the clause 14 restructuring that was incorporated in March of 1996. The prior wording was "the deduced template arguments are used to generate a single template function". This was changed to "deduced template arguments are used to instantiate a single function template specialization". I believe this resulted from what was basically a global replace of "generate" with "instantiate" and of "template function" with "function template specialization". In this case, the substitution changed the meaning. This paragraph needs reworking.

Proposed resolution (April, 2006):

Change 17.9.3 [temp.over] paragraph 1 as indicated:

...For each function template, if the argument deduction and checking succeeds, the *template-argument*s (deduced and/or explicit) are used to ~~instantiate~~ **synthesize the declaration of** a single function template specialization which is added to the candidate functions set to be used in overload resolution. If, for a given function template, argument deduction fails, no such function is added to the set of candidate functions for that template. The complete set of candidate functions includes all the ~~function templates instantiated in this way~~ **synthesized declarations** and all of the non-template overloaded functions of the

same name. The ~~function template specializations~~ **synthesized declarations** are treated like any other functions in the remainder of overload resolution, except as explicitly noted in 16.3.3 [over.match.best].

---

208. Rethrowing exceptions in nested handlers

Section: 18.1 [except.throw] **Status:** CD1 **Submitter:** Bill Wade **Date:** 28 Feb 2000

[Moved to DR at 4/01 meeting.]

Paragraph 7 of 18.1 [except.throw] discusses which exception is thrown by a *throw-expression* with no operand.

May an expression which has been "finished" (paragraph 7) by an inner catch block be rethrown by an outer catch block?

```
catch(...)    // Catch the original exception
{
  try{ throw; }    // rethrow it at an inner level
                   // (in reality this is probably
                   // inside a function)
  catch (...)
  {
  }   // Here, an exception (the original object)
      // is "finished" according to 15.1p7 wording

  // 15.1p7 says that only an unfinished exception
  // may be rethrown.
  throw;    // Can we throw it again anyway?  It is
            // certainly still alive (15.1p4).
}
```

I believe this is ok, since the paragraph says that the exception is finished when the "corresponding" catch clause exits. However since we have two clauses, and only one exception, it would seem that the one exception gets "finished" twice.

Proposed resolution (04/01):

1. In 18.1 [except.throw] paragraph 4, change

   > When the last handler being executed for the exception exits by any means other than `throw;` ...

   to

   > When the last remaining active handler for the exception exits by any means other than `throw;` ...

2. In 18.1 [except.throw] paragraph 6, change

   > A *throw-expression* with no operand rethrows the exception being handled.

   to

   > A *throw-expression* with no operand rethrows the currently handled exception (18.3 [except.handle]).

3. Delete 18.1 [except.throw] paragraph 7.

4. Add the following before 18.1 [except.throw] paragraph 6:

   > An exception is considered caught when a handler for that exception becomes active (18.3 [except.handle]). [*Note:* an exception can have active handlers and still be considered uncaught if it is rethrown.]

5. Change 18.3 [except.handle] paragraph 8 from

   > An exception is considered handled upon entry to a handler. [*Note:* the stack will have been unwound at that point.]

   to

   > A handler is considered active when initialization is complete for the formal parameter (if any) of the catch clause. [*Note:* the stack will have been unwound at that point.] Also, an implicit handler is considered active when `std::terminate()` or `std::unexpected()` is entered due to a throw. A handler is no longer considered active when the catch clause exits or when `std::unexpected()` exits after being entered due to a throw.
   >
   > The exception with the most recently activated handler that is still active is called the *currently handled exception*.

6. In 18.3 [except.handle] paragraph 16, change "exception being handled" to "currently handled exception."

---

428. Mention of expression with reference type

Section: 18.1 [except.throw] **Status:** CD1 **Submitter:** Steve Adamczyk **Date:** 14 July 2003

[Voted into WP at March 2004 meeting.]

18.1 [except.throw] paragraph 3 says that the type of a throw expression shall not be a pointer or reference to an incomplete type. But an expression never has reference type.

Proposed Resolution (October 2003):

Change the penultimate sentence of 18.1 [except.throw] paragraph 3 as follows:

> The type of the *throw-expression* shall not be an incomplete type, or a pointer ~~or reference~~ to an incomplete type **other than (possibly cv-qualified)** `void`~~, other than `void*`, `const void*`, `volatile void*`, or `const volatile void*`~~.

---

479. Copy elision in exception handling

Section: 18.1 [except.throw] **Status:** CD1 **Submitter:** Mike Miller **Date:** 07 Oct 2004

[Voted into WP at April, 2006 meeting.]

I have noticed a couple of confusing and overlapping passages dealing with copy elision. The first is 18.1 [except.throw] paragraph 5:

> If the use of the temporary object can be eliminated without changing the meaning of the program except for the execution of constructors and destructors associated with the use of the temporary object (15.2 [class.temporary]), then the exception in the handler can be initialized directly with the argument of the throw expression.

The other is 18.3 [except.handle] paragraph 17:

> If the use of a temporary object can be eliminated without changing the meaning of the program except for execution of constructors and destructors associated with the use of the temporary object, then the optional name can be bound directly to the temporary object specified in a *throw-expression* causing the handler to be executed.

I *think* these two passages are intended to describe the same optimization. However, as is often the case where something is described twice, there are significant differences. One is just different terminology — is "the exception in the handler" the same as "the object declared in the *exception-declaration* or, if the *exception-declaration* does not specify a name, a temporary object of that type" (18.3 [except.handle] paragraph 16)?

More significant, there is a difference in which kinds of *throw-expression*s are eligible for the optimization. In 18.1 [except.throw] paragraph 5, it appears that any object is a candidate, while in 18.3 [except.handle] paragraph 17 the thrown object must be a temporary ("the temporary object specified in a *throw-expression*"). For example, it's not clear looking at these two passages whether the copy of a local automatic can be elided. I.e., by analogy with the return value optimization described in 15.8 [class.copy] paragraph 15:

```
X x;
return x;    // copy may be elided

X x;
throw x;     // unclear whether copy may be elided
```

Which brings up another point: 15.8 [class.copy] paragraph 15 purports to be an exhaustive list in which copy elision is permitted even if the constructor and/or destructor have side effects; however, these two passages describe another case that is not mentioned in 15.8 [class.copy] paragraph 15.

A final point of confusion: in the unoptimized abstract machine, there are actually *two* copies in throwing and handling an exception: the copy from the object being thrown to the exception object, and the copy from the exception object to the object or temporary in the *exception-declaration*. 18.1 [except.throw] paragraph 5 speaks only of eliminating the exception object, copying the thrown object directly into the *exception-declaration* object, while 18.3 [except.handle] paragraph 17 refers to directly binding the *exception-declaration* object to the thrown object (if it's a temporary). Shouldn't these be separated, with a throw of an automatic object or temporary being like the return value optimization and the initialization of the object/temporary in the *exception-declaration* being a separate optimizable step (which could, presumably, be combined to effectively alias the *exception-declaration* onto the thrown object)?

(See paper J16/04-0165 = WG21 N1725.)

Proposed resolution (April, 2005):

1. Add two items to the bulleted list in 15.8 [class.copy] paragraph 15 as follows:

> This elision of copy operations is permitted in the following circumstances (which may be combined to eliminate multiple copies):
>
> - in a `return` statement in a function with a class return type, when the expression is the name of a non-volatile automatic object with the same cv-unqualified type as the function return type, the copy operation can be omitted by constructing the automatic object directly into the function's return value
> - **in a *throw-expression*, when the operand is the name of a non-volatile automatic object, the copy operation from the operand to the exception object (18.1 [except.throw]) can be omitted by constructing the automatic object directly into the exception object**
> - when a temporary class object that has not been bound to a reference (15.2 [class.temporary]) would be copied to a class object with the same cv-unqualified type, the copy operation can be omitted by constructing the temporary object directly into the target of the omitted copy

- **when the *exception-declaration* of an exception handler (clause 18 [except]) declares an object of the same type (except for cv-qualification) as the exception object (18.1 [except.throw]), the copy operation can be omitted by treating the *exception-declaration* as an alias for the exception object if the meaning of the program will be unchanged except for the execution of constructors and destructors for the object declared by the *exception-declaration***

2. Change 18.1 [except.throw] paragraph 5 as follows:

> ~~If the use of the temporary object can be eliminated without changing the meaning of the program except for the execution of constructors and destructors associated with the use of the temporary object (15.2 [class.temporary]), then the exception in the handler can be initialized directly with the argument of the throw expression.~~ When the thrown object is a class object, ~~and~~ the copy constructor ~~used to initialize the temporary copy is not~~ **and the destructor shall be** accessible, ~~the program is ill-formed (even when the temporary object could otherwise be eliminated)~~ **even if the copy operation is elided (15.8 [class.copy])**. ~~Similarly, if the destructor for that object is not accessible, the program is ill-formed (even when the temporary object could otherwise be eliminated).~~

3. Change 18.3 [except.handle] paragraph 17 as follows:

> ~~If the use of a temporary object can be eliminated without changing the meaning of the program except for execution of constructors and destructors associated with the use of the temporary object, then the optional name can be bound directly to the temporary object specified in a *throw-expression* causing the handler to be executed.~~ The copy constructor and destructor associated with the object shall be accessible even ~~when the temporary object is eliminated~~ **if the copy operation is elided (15.8 [class.copy])**.

---

592. Exceptions during construction of local static objects

Section: 18.2 [except.ctor] **Status:** CD1 **Submitter:** Alisdair Meredith **Date:** 30 August 2006

[Voted into the WP at the September, 2008 meeting (resolution in paper N2757).]

According to 18.2 [except.ctor] paragraph 2,

> An object that is partially constructed or partially destroyed will have destructors executed for all of its fully constructed subobjects, that is, for subobjects for which the principal constructor (15.6.2 [class.base.init]) has completed execution and the destructor has not yet begun execution. Similarly, if the non-delegating constructor for an object has completed execution and a delegating constructor for that object exits with an exception, the object's destructor will be invoked. Should a constructor for an element of an automatic array throw an exception, only the constructed elements of that array will be destroyed.

The requirement for destruction of array elements explicitly applies only to automatic arrays, and one might conclude from the context that only automatic class objects are in view as well, although that is not explicitly stated. What about local static arrays and class objects? Are they intended also to be subject to the requirement that fully-constructed subobjects are to be destroyed?

Proposed resolution (October, 2006):

Change 18.2 [except.ctor] paragraph 2 as follows:

> An object that is partially constructed or partially destroyed will have destructors executed for all of its fully constructed subobjects, that is, for subobjects for which the principal constructor (15.6.2 [class.base.init]) has completed execution and the destructor has not yet begun execution. Similarly, if the non-delegating constructor for an object has completed execution and a delegating constructor for that object exits with an exception, the object's destructor will be invoked. ~~Should a constructor for an element of an automatic array throw an exception, only the constructed elements of that array will be destroyed.~~ If the object or array was allocated in a *new-expression*, the matching deallocation function (6.7.4.2 [basic.stc.dynamic.deallocation], 8.3.4 [expr.new], 15.5 [class.free]), if any, is called to free the storage occupied by the object.

---

87. Exception specifications on function parameters

Section: 18.4 [except.spec] **Status:** CD1 **Submitter:** Steve Adamczyk **Date:** 25 Jan 1999

[Moved to DR at 4/01 meeting.]

In 18.4 [except.spec] paragraph 2:

> An exception-specification shall appear only on a function declarator in a function, pointer, reference or pointer to member declaration or definition.

Does that mean in the top-level function declarator, or one at any level? Can one, for example, specify an exception specification on a pointer-to-function parameter of a function?

```
void f(int (*pf)(float) throw(A))
```

Suggested answer: no. The exception specifications are valid only on the top-level function declarators.

However, if exception specifications are made part of a function's type as has been tentatively agreed, they would have to be allowed on any function declaration.

There is already an example of an exception specification for a parameter in the example in 18.4 [except.spec] paragraph 1.

Proposed resolution (04/01): Change text in 18.4 [except.spec] paragraph 1 from:

> An *exception-specification* shall appear only on a function declarator in a function, pointer, reference or pointer to member declaration or definition.

to:

> An *exception-specification* shall appear only on a function declarator for a function type, pointer to function type, reference to function type, or pointer to member function type that is the top-level type of a declaration or definition, or on such a type appearing as a parameter or return type in a function declarator.

(See also issues 25, 92, and 133.)

---

394. *identifier-list* is never defined

Section: 19 [cpp] **Status:** CD1 **Submitter:** Nicola Musatti **Date:** 16 Dec 2002

[Voted into WP at October 2004 meeting.]

In clause 19 [cpp], paragraph 1, the *control-line* non-terminal symbol is defined in terms of the *identifier-list* non-terminal, which is never defined within the standard document.

The same definition is repeated in clause A.14 [gram.cpp].

I suggest that the following definition is added to clause 19 [cpp], paragraph 1, after the one for *replacement-list*:

> *identifier-list*:
> *identifier*
> *identifier-list* , *identifier*

This should be repeated again in clause A.14 [gram.cpp], again after the one for *replacement-list*. It might also be desirable to include a third repetition in clause 19.3 [cpp.replace], paragraph 9.

Proposed Resolution (Clark Nelson, Dec 2003):

In clause 19 [cpp], paragraph 1, immediately before the definition of *replacement-list*, add:

> *identifier-list*:
> *identifier*
> *identifier-list* , *identifier*

If the correct TROFF macros are used, the definition will appear automatically in appendix A. It doesn't need to be repeated in 16.3p9.

With respect to the question of having the preprocessor description be synchronized with C99, this would fall into the category of a justified difference. (Other justified differences include those for Boolean expressions, alternative tokens, and terminology differences.)

---

370. Can `#include <...>` form be used other than for standard C++ headers?

Section: 19.2 [cpp.include] **Status:** CD1 **Submitter:** Beman Dawes **Date:** 01 August 2002

[Voted into WP at the October, 2006 meeting.]

The motivation for this issue is a desire to write portable programs which will work with any conforming implementation.

The C++ Standard (19.2 [cpp.include]) provides two forms of `#include` directives, with the <...> form being described (19.2 [cpp.include] paragraph 2) as "for a header", and the "..." form (19.2 [cpp.include] paragraph 3) as for "the source file" identified between the delimiters. When the standard uses the term "header", it often appears to be limiting the term to apply to the Standard Library headers only. Users of the standard almost always use the term "header" more broadly, to cover all `#include`d source files, but particularly those containing interface declarations.

Headers, including source files, can be categorized according to their origin and usage:

1. C++ Standard Library headers (which aren't necessarily files).
2. Other standard libraries such as the POSIX headers.
3. Operating system API's such as *windows.h*.
4. Third party libraries, such as Boost, ACE, or commercial offerings.
5. Organization-wide "standard" header files, such as a company's *config.hpp*.
6. A project's "public" header files, often shared by all developers working on the same project.

7. A project or user's "private", "local", or "detail" headers, in the same directory or sub-directory as the compilation unit.

Existing practice varies widely, but it is fairly easy to find users advocating:

- For (1), Standard Library headers, use <...>. All others use "...".
- For (7), use "...". All others use <...>.
- Draw a line somewhere below (1) and above (7), and use <...> above the line and "..." below. The exact location of the line may be based on a perception of how often the header involved changes; some "make" utilities use this to optimize builds.

Do any of the practices A, B, or C result in programs which can be rejected by a conforming implementation?

The first defect is that readers of the standard have not been able to reach consensus on the answers to the above question.

A second possible defect is that if A, B, or C can be rejected by a conforming implementation, then the standard should be changed because would mean there is a wide variance between the standard and existing practice.

Matt Austern: I really only see two positions:

1. Implementations have some unspecified mechanism (copying files to a magic directory, adding a magic flag,...) such that the line `#include <foo>` results in textual inclusion of the file foo.
2. Implementations are not required to have any mechanism for getting `#include <foo>` to perform textual inclusion of an arbitrary file `foo`. That form is reserved for standard library headers only.

I agree that the standard should clarify which of those two is the case (I imagine it'll hinge on finding one crucual sentence that either says "implementation defined" or "unspecified"), but from the standpoint of portability I don't see much difference between the two. I claim that, with either of those two interpretations, using `#include <foo>` is already nonportable.

(Of course, I claim that almost anything having to do with headers, including the `#include "foo"` form, is also nonportable. In practice there's wide variation in how compilers handle paths, especially relative paths.)

Beman Dawes: The whole issue can be resolved by replacing "header" with "header or source file" in 19.2 [cpp.include] paragraph 2. That will bring the standard into alignment with existing practice by both users and implementations. The "header and/or source file" wording is used at least three other places in the standard where otherwise there might be some confusion.

John Skaller: In light of Andrew Koenig's comments, this doesn't appear to be the case, since the mapping of include names to file names is implementation defined, and therefore source file inclusion cannot be made portable within the ISO C/C++ standards (since that provision obviously cannot be removed).

A possible idea is to create a binding standard, outside the C/C++ ISO Standards, which specifies not only the path lookup mechanism but also the translation from include names to file names. Clearly that is OS dependent, encoding dependent, etc, but there is no reason not to have a binding standard for Unix, Windows, etc, and specify these bindings in such a way that copying directories from one OS to the other can result in programs working on both OS's.

Andy Koenig: An easier solution might be to specify a (presumably unbounded, or bounded only by implementation capacity) collection of header-file names that every implementation must make it possible for programs to access somehow, without specifying exactly how.

Notes from October 2002 meeting:

This was discussed at some length. While there was widespread agreement that such inclusion is inherently implementation-dependent, we agreed to try to add wording that would make it clear that implementations are permitted (but not required) to allow inclusion of files using the <...> form of `#include`.

Proposed resolution (April, 2005):

Change 19.2 [cpp.include] paragraph 7 from:

> [*Example:* The most common uses of `#include` preprocessing directives are as in the following:
>
> ```
> #include <stdio.h>
> #include "myprog.h"
> ```
>
> —*end example*]

to:

> [*Note:* Although an implementation may provide a mechanism for making arbitrary source files available to the < > search, in general programmers should use the < > form for headers provided with the implementation, and the " " form for sources outside the control of the implementation. For instance:
>
> ```
> #include <stdio.h>
> #include <unistd.h>
> #include "usefullib.h"
> #include "myprog.h"
> ```
>
> —*end note*]

Notes from October, 2005 meeting:

Some doubt was expressed as to whether the benefit of this non-normative clarification outweighs the overall goal of synchronizing clause 16 with the corresponding text in the C99 Standard. As a result, this issue is being left in "review" status to allow further discussion.

Additional notes (October, 2006):

WG14 takes no position on this change.

---

106. Creating references to references during template deduction/instantiation

Section: unknown [unknown] **Status:** CD1 **Submitter:** Bjarne Stroustrup **Date:** unknown

[Moved to DR at 10/01 meeting.]

The main defect is in the library, where the binder template can easily lead to reference-to-reference situations.

Proposed resolution (04/01):

1. Add the following as paragraph 6 of 10.1.3 [dcl.typedef]:

   > If a typedef `TD` names a type "reference to *cv1* `S`," an attempt to create the type "reference to *cv2* `TD`" creates the type "reference to *cv12* `S`," where *cv12* is the union of the cv-qualifiers *cv1* and *cv2*. Redundant qualifiers are ignored. [*Example:*
   >
   > ```
   > int i;
   > typedef int& RI;
   > RI& r = i;          // r has the type int&
   > const RI& r = i;    // r has the type const int&
   > ```
   >
   > —*end example*]

2. Add the following as paragraph 4 of 17.3.1 [temp.arg.type]:

   > If a *template-argument* for a *template-parameter* `T` names a type "reference to *cv1* `S`," an attempt to create the type "reference to *cv2* `T`" creates the type "reference to *cv12* `S`," where *cv12* is the union of the cv-qualifiers *cv1* and *cv2*. Redundant cv-qualifiers are ignored. [*Example:*
   >
   > ```
   > template <class T> class X {
   >     f(const T&);
   >     /* ... */
   > };
   > X<int&> x;    // X<int&>::f has the parameter type const int&
   > ```
   >
   > —*end example*]

3. In 17.9.2 [temp.deduct] paragraph 2 bullet 3 sub-bullet 5, remove the indicated text:

   > Attempting to create ~~a reference to a reference type or~~ a reference to `void`.

(See also paper J16/00-0022 = WG21 N1245.)

Issues with "CD2" Status

---

816. Diagnosing violations of `[[final]]`

Section: _N3225_.7.6.4 [dcl.attr.final] **Status:** CD2 **Submitter:** UK **Date:** 3 March, 2009

N2800 comment UK 108

[Voted into WP at July, 2009 meeting.]

According to _N3225_.7.6.4 [dcl.attr.final] paragraph 2, overriding a virtual function with the `[[final]]` attribute renders a program ill-formed, but no diagnostic is required. This is easily diagnosable and a diagnostic should be required in this case.

Notes from the March, 2009 meeting:

This specification was a deliberate decision on the part of the EWG; the general rule was that it should be possible to ignore attributes without changing the meaning of a program. However, the consensus of the CWG was that violation of the `[[final]]` attribute should require a diagnostic.

Proposed resolution (March, 2009):

Change _N3225_.7.6.4 [dcl.attr.final] paragraph 2 as follows:

> If a virtual member function `f` in some class `B` is marked `final` and in a class `D` derived from `B` a function `D::f` overrides `B::f`, the program is ill-formed~~; no diagnostic required. [*Footnote:* If an implementation does not emit a diagnostic it should execute the program as if final were not present. —*end footnote*]~~

---

817. Meaning of `[[final]]` applied to a class definition

Section: _N3225_.7.6.4 [dcl.attr.final] **Status:** CD2 **Submitter:** US **Date:** 3 March, 2009

N2800 comment US 42

[Voted into WP at March, 2010 meeting.]

According to _N3225_.7.6.4 [dcl.attr.final] paragraph 1, the `[[final]]` attribute applied to a class is just a shorthand notation for marking each of the class's virtual functions as `[[final]]`. This is different from the similar usage in other languages, where it means that the class so marked cannot be used as a base class. This discrepancy is confusing, and the definition used by the other languages is more useful.

Notes from the March, 2009 meeting:

The intent of the `[[final]]` attribute is as an aid in optimization, to avoid virtual function calls when the final overrider is known. It is possible to use the `[[final]]` attribute to prevent derivation by marking the destructor as `[[final]]`; in fact, as most polymorphic classes will, as a matter of good programming practice, have a virtual destructor, marking the class as `[[final]]` will have the effect of preventing derivation.

Nonetheless, the general consensus of the CWG was to change the meaning of class `[[final]]` to parallel the usage in other languages.

Proposed resolution (October, 2009):

1. Change _N3225_.7.6.4 [dcl.attr.final] paragraph 1 and add a new paragraph, as follows:

   > The *attribute-token* `final` specifies **derivation semantics for a class and** overriding semantics for a virtual function. It shall appear at most once in each *attribute-list* and no *attribute-argument-clause* shall be present. The attribute applies to class definitions and to virtual member functions being declared in a class definition. ~~If the attribute is specified for a class definition, it is equivalent to being specified for each virtual member function of that class, including inherited member functions.~~
   >
   > **If some class `B` is marked `final` and a class `D` is derived from `B` the program is ill-formed.**

2. Change the example in _N3225_.7.6.4 [dcl.attr.final] paragraph 3 as follows:

```
struct B1 {
  virtual void f [[ final ]] ();
};

struct D1 : B1 {
  void f();          // ill-formed
};

struct [[ final ]] B2 {
};

struct D2 : B2 {     // ill-formed
};
```

---

789. Deprecating trigraphs

Section: _N4140_.2.4 [lex.trigraph] **Status:** CD2 **Submitter:** UK **Date:** 3 March, 2009

N2800 comment UK 11

[Voted into WP at March, 2010 meeting as document N3077.]

Trigraphs are a complicated solution to an old problem, that cause more problems than they solve in the modern environment. Unexpected trigraphs in string literals and occasionally in comments can be very confusing for the non-expert. They should be deprecated.

Notes from the March, 2009 meeting:

IBM, at least, uses trigraphs in its header files in conditional compilation directives to select character-set dependent content in a character-set independent fashion and would thus be negatively affected by the removal of trigraphs. One possibility that was discussed was to avoid expanding trigraphs inside character string literals, which is the context that causes most surprise and confusion, but still to support them in the rest of the program text. Specifying that approach, however, would be challenging because trigraphs are replaced in phase 1, before character strings are recognized in phase 3. See also the similar discussion of universal-character-names in issue 787.

The consensus of the CWG was that trigraphs should be deprecated.

Proposed resolution (September, 2009):

See paper PL22.16/09-0168 = WG21 N2978.

Notes from the October, 2009 meeting:

The CWG is interested in exploring other alternatives that address the particular problem of trigraphs in raw strings but that do not require the grammar changes of the approach in N2978. One possibility might be to recognize raw strings in some way in translation phase 1.

Notes from the March, 2010 meeting:

The CWG decided not to deprecate trigraphs, acknowledging that there are communities in which they are viewed as necessary. Instead, it was decided to address what was considered to be the most pressing issue regarding trigraphs, that is, recognizing trigraph sequences inside raw string literals.

---

743. Use of `decltype` in a *nested-name-specifier*

Section: _N4567_.5.1.1 [expr.prim.general] **Status:** CD2 **Submitter:** Jaakko Järvi **Date:** 12 November, 2008

N2800 comment JP 8

[Voted into WP at March, 2010 meeting as document N3049.]

The grammar for *nested-name-specifier* in _N4567_.5.1.1 [expr.prim.general] paragraph 7 does not allow `decltype` to be used in a *qualified-id*. This could be useful for cases like:

```
auto vec = get_vec();
decltype(vec)::value_type v = vec.first();
```

(See also issue 950.)

Proposed resolution (September, 2009):

See paper PL22.16/09-0181 = WG21 N2991.

Proposed resolution (February, 2010):

See paper PL22.16/10-0021 = WG21 N3031.

---

760. `this` inside a nested class of a non-static member function

Section: _N4567_.5.1.1 [expr.prim.general] **Status:** CD2 **Submitter:** Mike Miller **Date:** 3 February, 2009

[Voted into WP at March, 2010 meeting.]

`this` is a keyword and thus not subject to ordinary name lookup. That makes the interpretation of examples like the following somewhat unclear:

```
struct outer {
  void f() {
    struct inner {
      int a[sizeof(*this)];  // #1
    };
  }
};
```

According to _N4567_.5.1.1 [expr.prim.general] paragraph 3,

> The keyword `this` shall be used only inside a non-static class member function body (12.2.1 [class.mfct]) or in a *brace-or-equal-initializer* for a non-static data member.

Should the use of `this` at #1 be interepreted as a well-formed reference to `outer::f()`'s `this` or as an ill-formed attempt to refer to a `this` for `outer::inner`?

One possible interpretation is that the intent is as if `this` were an ordinary identifier appearing as a parameter in each non-static member function. (This view applies to the initializers of non-static data members as well if they are considered to be rewritten as *mem-initializer*s in the constructor body.) Under this interpretation, the prohibition against using `this` in other contexts simply falls out of the fact that name lookup would fail to find `this` anywhere else, so the reference in the example is well-formed. (Implementations vary in their treatment of this example, so clearer wording is needed, whichever way the interpretation goes.)

Proposed resolution (February, 2010):

Change _N4567_.5.1.1 [expr.prim.general] paragraph 2 as follows:

> ...The keyword `this` shall be used only inside **the body of** a non-static ~~class~~ member function ~~body~~ (12.2.1 [class.mfct]) **of the nearest enclosing class** or in a *brace-or-equal-initializer* for a non-static data member **(12.2 [class.mem])**. The type of the expression is a pointer to the class of the function or non-static data member, possibly with cv-qualifiers on the class type. The expression is an rvalue. **[*Example:***

```
class Outer {
  int a[sizeof(*this)];             // error: not inside a member function
  unsigned int sz = sizeof(*this); // OK, in brace-or-equal-initializer

  void f() {
    int b[sizeof(*this)];           // OK

    struct Inner {
      int c[sizeof(*this)];         // error: not inside a member function of Inner
    };
  }
};
```

—end example]

---

850. Restrictions on use of non-static data members

Section: _N4567_.5.1.1 [expr.prim.general] **Status:** CD2 **Submitter:** Jason Merrill **Date:** 1 April, 2009

[Voted into WP at October, 2009 meeting.]

The resolution of issue 613, as reflected in the sixth bullet of _N4567_.5.1.1 [expr.prim.general] paragraph 10, allows an *id-expression* designating a non-static data member to be used

- if... it is the sole constituent of an unevaluated operand, except for optional enclosing parentheses.

The requirement that the *id-expression* be the "sole constituent" of the unevaluated operand seems unnecessarily strict, forbidding such plausible use cases as

```
struct S {
    int ar[42];
};
int i = sizeof(S::ar[0]);
```

or the use of the member as a function argument in template metaprogramming. The more general version of the restriction seems not to be very difficult to implement and may actually represent a simplification in some implementations.

Proposed resolution (July, 2009):

Change _N4567_.5.1.1 [expr.prim.general] paragraph 10 as follows:

- ...
- if that *id-expression* denotes a non-static data member and it ~~is the sole constituent of~~ **appears in** an unevaluated operand~~, except for optional enclosing parentheses~~. [*Example:*

```
struct S {
  int m;
};
int i = sizeof(S::m);         // OK
int j = sizeof(S::m + 42);    // ~~error: reference to non-static member in subexpression~~ OK
```

—end example]

---

690. The dynamic type of an rvalue reference

Section: 3 [intro.defs] **Status:** CD2 **Submitter:** Eelis van der Weegen **Date:** 7 April, 2008

N2800 comment FR 5

[Voted into WP at March, 2010 meeting as document N3055.]

According to 3 [intro.defs], "dynamic type,"

> The dynamic type of an rvalue expression is its static type.

This is not true of an rvalue reference, which can be bound to an object of a class type derived from the reference's static type.

Proposed resolution (June, 2008):

Change 3 [intro.defs], "dynamic type," as follows:

> the type of the most derived object (4.5 [intro.object]) to which ~~the lvalue denoted by~~ an lvalue **or an rvalue-reference (clause 8 [expr])** expression refers. [*Example:* if a pointer (11.3.1 [dcl.ptr]) `p` whose static type is "pointer to class `B`" is pointing to an object of class `D`, derived from `B` (clause 13 [class.derived]), the dynamic type of the expression `*p` is "D." References (11.3.2 [dcl.ref]) are treated similarly. *—end example*] The dynamic type of an rvalue expression **that is not an rvalue reference** is its static type.

Notes from the June, 2008 meeting:

Because expressions have an rvalue reference type only fleetingly, immediately becoming either lvalues or rvalues and no longer references, the CWG expressed a desire for a different approach that would somehow describe an rvalue that resulted from an rvalue reference instead of using the concept of an expression that is an rvalue reference, as above. This approach could also be used in the resolution of issue 664.

Additional note (August, 2008):

This issue, along with issue 664, indicates that rvalue references have more in common with lvalues than with other rvalues: they denote particular objects, thus allowing object identity and polymorphic behavior. That suggests that these issues may be just the tip of the iceberg: restrictions on out-of-lifetime access to objects, the aliasing rules, and many other specifications are written to apply only to lvalues, on the assumption that only lvalues refer to specific objects. That assumption is no longer valid with rvalue references.

This suggests that it might be better to classify all rvalue references, not just named rvalue references, as lvalues instead of rvalues, and then just change the reference binding, overload resolution, and template argument deduction rules to cater to the specific kind of lvalues that are associated with rvalue references.

Additional note, May, 2009:

Another place in the Standard where the assumption is made that only lvalues can have dynamic types that differ from their static types is 8.2.8 [expr.typeid] paragraph 2.

(See also issues 846 and 863.)

Additional note, September, 2009:

Yet another complication is the statement in 6.10 [basic.lval] paragraph 9 stating that "non-class rvalues always have cv-unqualified types." If an rvalue reference is an rvalue, then the following example is well-formed:

```
void f(int&&);      // reference to non-const
void g() {
    const int i = 0;
    f(static_cast<const int&&>(i));
}
```

The `static_cast` binds an rvalue reference to the const object `i`, but the fact that it's an rvalue means that the cv-qualification is lost, effectively allowing the parameter of `f`, a reference to non-const, to bind directly to the const object.

Proposed resolution (February, 2010):

See paper N3030.

---

612. Requirements on a conforming implementation

Section: 4.6 [intro.execution] **Status:** CD2 **Submitter:** Clark Nelson **Date:** 23 January 2007

[Voted into WP at October, 2009 meeting.]

The execution requirements on a conforming implementation are described twice in the Standard, once in 4.6 [intro.execution] paragraphs 5-6 and again in paragraph 11. These descriptions differ in at least a couple of important ways:

The most significant discrepancy has to do with the way output is described. In paragraph 11, the least requirements are described in terms of data written at program termination, clearly allowing arbitrary buffering, whereas in paragraph 6, the observable behavior is described in terms of calls to I/O functions. For example, there are compilers which transform a call to printf with a single argument into a call to fputs. That's valid under paragraph 11, but not under paragraph 6.

Also, in paragraph 6, volatile accesses and I/O operations are included in a single sequence, suggesting that they are equally constrained by sequencing requirements, whereas in paragraph 11, they are clearly not.

There are also editorial discrepancies that should be cleaned up.

Proposed resolution (September, 2009):

The resolution of issue 785 also resolves this issue.

---

785. "Execution sequence" is inappropriate phraseology

Section: 4.6 [intro.execution] **Status:** CD2 **Submitter:** US/UK **Date:** 3 March, 2009

N2800 comment US 16
N2800 comment UK 8
N2800 comment UK 7

[Voted into WP at October, 2009 meeting.]

In the presence of threads, it is no longer appropriate to characterize the abstract machine as having an "execution sequence."

Proposed resolution (September, 2009):

1. Change 4.6 [intro.execution] paragraph 3 as follows:

   ...An instance of the abstract machine can thus have more than one possible execution ~~sequence~~ for a given program and a given input.

2. Change 4.6 [intro.execution] paragraph 5 as follows:

   A conforming implementation executing a well-formed program shall produce the same observable behavior as one of the possible ~~execution sequences~~ **executions** of the corresponding instance of the abstract machine with the same program and the same input. However, if any such execution ~~sequence~~ contains an undefined operation, this International Standard places no requirement on the implementation executing that program with that input (not even with regard to operations preceding the first undefined operation).

3. Delete 4.6 [intro.execution] paragraph 6, including the footnote:

   ~~The observable behavior of the abstract machine is its sequence of reads and writes to `volatile` data and calls to library I/O functions. [*Footnote:* An implementation can offer additional library I/O functions as an extension. Implementations that do so should treat calls to those functions as "observable behavior" as well. —*end footnote*]~~

4. Change 4.6 [intro.execution] paragraph 9 as follows:

   The least requirements on a conforming implementation are:

   - Access to volatile objects are evaluated strictly according to the rules of the abstract machine.
   - At program termination, all data written into files shall be identical to one of the possible results that execution of the program according to the abstract semantics would have produced.
   - The input and output dynamics of interactive devices shall take place in such a fashion that ~~prompting messages actually appear prior to a program waiting~~ **prompting output is actually delivered before a program waits** for input. What constitutes an interactive device is implementation-defined.

   These collectively are referred to as the observable behavior of the program. [*Note:* more stringent correspondences between abstract and actual semantics may be defined by each implementation. —*end note*]

(Note; this resolution also resolves issue 612.)

---

726. Atomic and non-atomic objects in the memory model

Section: 4.7 [intro.multithread] **Status:** CD2 **Submitter:** Clark Nelson **Date:** 30 September, 2008

[Voted into WP at October, 2009 meeting.]

In general, the description of the memory model is very careful to specify when the objects under discussion are atomic or non-atomic. However, there are a few cases where it could be clearer.

Proposed resolution (March, 2009):

1. Modify 4.7 [intro.multithread] paragraph 5 as follows:

   All modifications to a particular atomic object *M* occur in some particular total order, called the *modification order of M*. If *A* and *B* are modifications of an atomic object *M* and *A* happens before (as defined below) *B*, then *A* shall precede *B* in the modification order of *M*, which is defined below. [*Note:* This states that the modification orders must respect *happens before*. —*end note*] [*Note:* There is a separate order for each ~~scalar~~ **atomic** object. There is no requirement that these can be combined into a single total order for all objects. In general this will be impossible since different threads may observe modifications to different variables in inconsistent orders. —*end note*]

2. Modify 4.7 [intro.multithread] paragraph 7 as follows:

   Certain library calls *synchronize with* other library calls performed by another thread. In particular, an atomic operation *A* that performs a release operation on an **atomic** object *M* synchronizes with an atomic operation *B* that performs an acquire operation on *M* and reads a value written by any side effect in the release sequence headed by *A*...

3. Modify 4.7 [intro.multithread] paragraph 12 as follows:

   A *visible side effect A* on ~~an~~ **a scalar** object **or bit-field** *M* with respect to a value computation *B* of *M* satisfies the conditions:

   - *A* happens before *B*, and
   - there is no other side effect *X* to *M* such that *A* happens before *X* and *X* happens before *B*.

The value of a non-atomic scalar object **or bit-field** *M*, as determined by evaluation *B*, shall be the value stored by the visible side effect *A*. [*Note:* If there is ambiguity about which side effect to a non-atomic object **or bit-field** is visible, then there is a data race, and the behavior is undefined. —*end note*] ...

---

740. Incorrect note on data races

Section: 4.7 [intro.multithread] **Status:** CD2 **Submitter:** Wolf Lammen **Date:** 3 November, 2008

[Voted into WP at March, 2010 meeting.]

4.7 [intro.multithread] paragraph 12 says,

> A *visible side effect A* on an object *M* with respect to a value computation *B* of *M* satisfies the conditions:
>
> - *A* happens before *B*, and
> - there is no other side effect *X* to *M* such that *A* happens before *X* and *X* happens before *B*.
>
> The value of a non-atomic scalar object *M*, as determined by evaluation *B*, shall be the value stored by the visible side effect *A*. [*Note:* If there is ambiguity about which side effect to a non-atomic object is visible, then there is a data race, and the behavior is undefined. —*end note*]

The note here suggests that, except in the case of a data race, visible side effects to value computation can always be determined. But unsequenced and indeterminately sequenced side effects on the same object create ambiguities with respect to a later value computation as well. So the wording needs to be revisited, see the following examples.

```
int main(){
  int i = 0;
  i = // unsequenced side effect A
  i++; // unsequenced side effect B
  return i; // value computation C
}
```

According to the definition in the draft, both A and B are visible side effects to C. However, there is no data race, because (paragraph 14) a race involves at least two threads. So the note in paragraph 12 is logically false.

The model introduces the special case of indeterminately sequenced side effects, that leave open what execution order is taken in a concrete situation. If the execution paths access the same data, unpredictable results are possible, just as it is the case with data races. Whereas data races constitute undefined behavior, indeterminatedly sequenced side effects on the same object do not. As a consequence of this disparity, indeterminately sequenced execution occasionally needs exceptional treatment.

```
int i = 0;
int f(){
  return
  i = 1; // side effect A
}
int g(){
  return
  i = 2; // side effect B
}
int h(int, int){
  return i; // value computation C
}
int main(){
  return h(f(),g()); // function call D returns 1 or 2?
}
```

Here, either A or B is the visible side effect on the value computation C, but you cannot tell which (cf. 4.6 [intro.execution] paragraph 16). Although an ambiguity is present, it is neither because of a data race, nor is the behavior undefined, in total contradiction to the note.

Proposed resolution (October, 2009):

Change 4.7 [intro.multithread] paragraph 12 as follows:

> ...The value of a non-atomic scalar object or bit-field *M*, as determined by evaluation *B*, shall be the value stored by the visible side effect *A*. [*Note:* If there is ambiguity about which side effect to a non-atomic object or bit-field is visible, then ~~there is a data race, and~~ the behavior is **either unspecified or** undefined. —*end note*]...

---

786. Definition of "thread"

Section: 4.7 [intro.multithread] **Status:** CD2 **Submitter:** US **Date:** 3 March, 2009

N2800 comment US 17

[Voted into WP at October, 2009 meeting.]

The term "thread" is introduced but not defined in 4.7 [intro.multithread] paragraph 1. A definition is needed.

Proposed resolution (September, 2009):

Chamge 4.7 [intro.multithread] paragraph 1 as follows:

> **A *thread of execution* (a.k.a. *thread*) is a single flow of control within a program, including the initial invocation of a specific top-level function, and recursively including every function invocation subsequently executed by the thread. [*Note:* When one thread creates another, the initial call to the top-level function of the new thread is executed by the new thread, not by the creating thread. —*end note*] Every thread in a program can potentially access every object and function in the program. [*Footnote:* An object with automatic or thread storage duration (6.7 [basic.stc]) is associated with one specific thread, and can be accessed by a different thread only indirectly through a pointer or reference (6.9.2 [basic.compound]). —*end footnote*]** Under a hosted implementation, a C++ program can have more than one ~~*thread of execution* (a.k.a. *thread*)~~ **thread** running concurrently...

---

787. Unnecessary lexical undefined behavior

Section: 5.2 [lex.phases] **Status:** CD2 **Submitter:** UK **Date:** 3 March, 2009

N2800 comment UK 9
N2800 comment UK 12

[Voted into WP at March, 2010 meeting.]

There are several instances of undefined behavior in lexical processing:

- 5.2 [lex.phases] paragraph 1, phase 2: a universal-character-name resulting from a line splice.
- 5.2 [lex.phases] paragraph 1, phase 2: a file ending without a new-line character or with a new-line character that is spliced away.
- 5.2 [lex.phases] paragraph 1, phase 4: a universal-character-name resulting from macro token concatenation.
- 5.8 [lex.header] paragraph 2: `'`, `\`, `/*`, `//`, or `"` appearing in a *header-name*.

These would be more appropriately handled as conditionally-supported behavior, requiring implementations either to document their handling of these constructs or to issue a diagnostic.

Additional note, March, 2009:

The undefined behavior referred to above regarding universal-character-names is the result of the considerations described in the C99 Rationale, section 5.2.1, in the part entitled "UCN models." Three different models for support of UCNs are described, each involving different conversions between UCNs and wide characters and/or at different times during program translation. Implementations, as well as the specification in a language standard, can employ any of the three, but it must be impossible for a well-defined program to determine which model was actually employed by implementation. The implication of this "equivalence principle" is that any construct that would give different results under the different models must be classified as undefined behavior. For example, an apparent UCN resulting from a line-splice would be recognized as a UCN by an implementation in which all wide characters were translated immediately into UCNs, as described in C++ phase 1, but would not be recognized as a UCN by another implementation in which all UCNs were translated immediately into wide characters (a possibility mentioned parenthetically in C++ phase 1).

There are additional implications for this "equivalence principle" beyond the ones identified in the UK CD comments. See also issue 578; presumably a string like the one in that issue should also be described as having undefined behavior. Also, because C++'s model introduces backslash characters as part of UCNs for any character outside the basic source character set, any *header-name* that contains such a character (e.g., `#include "@.h"`) will have undefined behavior in C++. This is also the reason that UCNs are translated into wide characters inside raw strings: two of the three models articulated in the C99 Rationale translate to or from UCNs in phase 1, before raw strings are recognized as tokens in phase 3, so raw strings cannot treat UCNs differently from the way they are treated in other contexts. See also issue 789 for similar points regarding trigraphs.

Notes from the October, 2009 meeting:

The CWG decided that the non-UCN aspects of this issue should be resolved, while the overall questions regarding trigraphs, UCNs, and raw strings will be investigated separately.

Proposed resolution (February, 2010):

1. Change 5.2 [lex.phases] paragraph 1 phase 2 as follows:

   > ...~~If a~~ **A** source file that is not empty **and that** does not end in a new-line character, or **that** ends in a new-line character immediately preceded by a backslash character before any such splicing takes place, ~~the behavior is undefined~~ **shall be processed as if an additional new-line character were appended to the file**.

2. Change 5.8 [lex.header] paragraph 2 as follows:

   > ~~If~~ **The appearance of** either of the characters `'` or `\`~~,~~ or **of** either of the character sequences `/*` or `//` ~~appears~~ in a *q-char-sequence* or ~~a~~ **an** *h-char-sequence* **is conditionally-supported with implementation-defined semantics**, ~~or~~ **as is the appearance of** the character `"` ~~appears~~ in ~~a~~ **an** *h-char-sequence*~~, the behavior is undefined~~. [*Footnote:* Thus, **a** sequence~~s~~ of characters that resemble**s an** escape sequence~~s cause undefined behavior~~ **might result in an error, be interpreted as**

the character corresponding to the escape sequence, or have a completely different meaning, depending on the implementation. —*end footnote*]

---

630. Equality of narrow and wide character values in the basic character set

Section: 5.3 [lex.charset] **Status:** CD2 **Submitter:** Tom Plum **Date:** 21 April 2007

[Voted into WP at October, 2009 meeting.]

WG14 accepted DR 279 regarding the rule known colloquially as the `L'x'=='x'` rule. This change was made to C99 in TC2. The Austin Group subsequently opened DR 321 against TC2, observing that the change made in TC2 would invalidate existing conforming C code that relied on the `L'x'=='x'` rule.

DR 321 is now closed and will be included in the CD3 to C99. This change defines a new standard macro, which WG14 drafted as follows:

> `__STDC_MB_MIGHT_NEQ_WC__`: The integer constant `1`, intended to indicate that there might be some character `x` in the basic character set, such that `'x'` need not be equal to `L'x'`.

WG14 requests that WG21 adopt this revision and this macro in C++0x.

Proposed resolution (July, 2009):

Add the following to 19.8 [cpp.predefined] paragraph 2:

> `__STDC_MB_MIGHT_NEQ_WC__`
> **The integer constant `1`, intended to indicate that, in the encoding for `wchar_t`, a member of the basic character set need not have a code value equal to its value when used as the lone character in an ordinary character literal.**

---

788. Relationship between locale and values of the execution character set

Section: 5.3 [lex.charset] **Status:** CD2 **Submitter:** FR **Date:** 3 March, 2009

N2800 comment FR 10

[Voted into WP at March, 2010 meeting.]

According to 5.3 [lex.charset] paragraph 3,

> The values of the members of the execution character sets are implementation-defined, and any additional members are locale-specific.

This makes it sound as if the locale determines only whether an extended character (one not in the basic execution character set) exists, not its value (which is just implementation-defined, not locale-specific). The description should be clarified to indicate that the value of a given character can vary between locales, as well.

Proposed resolution (February, 2010):

Change 5.3 [lex.charset] paragraph 3 as follows:

> ...The *execution character set* and the *execution wide-character set* are **implementation-defined** supersets of the basic execution character set and the basic execution wide-character set, respectively. The values of the members of the execution character sets **and the sets of additional members** ~~are implementation-defined, and any additional members~~ are locale-specific.

---

832. Value of preprocessing numbers

Section: 5.9 [lex.ppnumber] **Status:** CD2 **Submitter:** UK **Date:** 3 March, 2009

N2800 comment UK 13

[Voted into WP at October, 2009 meeting.]

5.9 [lex.ppnumber] paragraph 2 says,

> A preprocessing number does not have a type or a value; it acquires both after a successful conversion (as part of translation phase 7, 5.2 [lex.phases]) to an integral literal token or a floating literal token.

However, preprocessing directives are executed in phase 4, and the evaluation of *constant-expression*s in `#if` directives requires that preprocessing numbers have values.

Proposed resolution (July, 2009):

Change 5.9 [lex.ppnumber] paragraph 2 as follows:

> A preprocessing number does not have a type or a value; it acquires both after a successful conversion ~~(as part of translation phase 7 (5.2 [lex.phases]))~~ to an integral literal token or a floating literal token.

---

933. 32-bit UCNs with 16-bit `wchar_t`

Section: 5.13.3 [lex.ccon] **Status:** CD2 **Submitter:** Alisdair Meredith **Date:** 7 July, 2009

[Voted into WP at October, 2009 meeting.]

According to 5.13.3 [lex.ccon] paragraph 2,

> A character literal that begins with the letter `L`, such as `L'x'`, is a wide-character literal. A wide-character literal has type `wchar_t`. The value of a wide-character literal containing a single *c-char* has value equal to the numerical value of the encoding of the *c-char* in the execution wide-character set.

A *c-char* that is a universal character name might, when translated to the execution character set, result in a multi-character sequence that is larger than can be represented in a `wchar_t`. There is wording that prevents this in `char16_t` literals, but not for `wchar_t` literals. This seems undesirable.

Proposed resolution (July, 2009):

1. Change 5.13.3 [lex.ccon] paragraph 2 as follows:

> ...The value of a wide-character literal containing a single *c-char* has value equal to the numerical value of the encoding of the *c-char* in the execution wide-character set**, unless the *c-char* has no representation in the execution wide-character set, in which case the value is implementation-defined. [*Note:* The type `wchar_t` is able to represent all members of the execution wide-character set, see 6.9.1 [basic.fundamental]. —*end note*]**. The value of a wide-character literal containing multiple *c-char*s is implementation-defined.

2. Change 5.13.3 [lex.ccon] paragraph 5 as follows:

> A universal-character-name is translated to the encoding, in the **appropriate** execution character set, of the character named...

---

790. Concatenation of raw and non-raw string literals

Section: 5.13.5 [lex.string] **Status:** CD2 **Submitter:** JP **Date:** 3 March, 2009

N2800 comment JP 5

[Voted into WP at October, 2009 meeting.]

The description of concatenation of string literals in 5.13.5 [lex.string] paragraph 11 does not mention raw strings explicitly, so it is not clear whether, and if so, how, they combine with non-raw strings.

Notes from the March, 2009 meeting:

A raw string should be considered equivalent to the corresponding non-raw string in string literal concatenation.

Proposed resolution (September, 2009):

1. In 5.13.5 [lex.string], replace the definition of *string-literal* with:

> *string-literal:*
> *encoding-prefix*$_{opt}$ `"` *s-char-sequence*$_{opt}$ `"`
> *encoding-prefix*$_{opt}$ `R` *raw-string*
>
> *encoding-prefix:*
> `u8`
> `u`
> `U`
> `L`

2. Change 5.13.5 [lex.string] paragraph 5 as follows:

> ~~A~~ **After translation phase 6, a** string literal that does not begin with ~~`u8`, `u`, `U`, or `L`~~ **an *encoding-prefix*** is an ordinary string literal, and is initialized with the given characters.

3. Change 5.13.5 [lex.string] paragraph 12 as follows:

In translation phase 6 (5.2 [lex.phases]), adjacent string literals are concatenated. If both string literals have the same ~~prefix~~ ***encoding-prefix***, the resulting concatenated string literal has that ~~prefix~~ ***encoding-prefix***. If one string literal has no ~~prefix~~ ***encoding-prefix***, it is treated as a string literal of the same ~~prefix~~ ***encoding-prefix*** as the other operand. If a UTF-8 string literal token is adjacent to a wide string literal token, the program is ill-formed. Any other concatenations are conditionally supported with implementation-defined behavior. [*Note:* This concatenation is an interpretation, not a conversion. **Because the interpretation happens in translation phase 6 (after each character from each literal has been translated into a value from the appropriate character set), a string literal's initial rawness has no effect on the interpretation or well-formedness of the concatenation.** —*end note*] [*Example*:...

(Note: this resolution also resolves issue 834.)

---

834. What is an "ordinary string literal" ?

Section: 5.13.5 [lex.string] **Status:** CD2 **Submitter:** Mike Miller **Date:** 6 March, 2009

[Voted into WP at October, 2009 meeting.]

According to 5.13.5 [lex.string] paragraph 4,

> A string literal that does not begin with `u8`, `u`, `U`, or `L` is an ordinary string literal, and is initialized with the given characters.

This is not as clear as it could be that a string like `u8R"[xxx]"` is not an ordinary string literal, because the string's prefix is not one of those listed (i.e., it's not obvious that possible substrings of the prefix are in view). This would be clearer if it simply said,

> A string literal with no prefix or a prefix of `R` is an ordinary string literal.

Proposed resolution (September, 2009):

This issue is resolved by the resolution of issue 790.

---

872. Lexical issues with raw strings

Section: 5.13.5 [lex.string] **Status:** CD2 **Submitter:** Joseph Myers **Date:** 16 April, 2009

[Voted into WP at March, 2010 meeting as document N3077.]

The specification of raw string literals interacts poorly with the specification of preprocessing tokens. The grammar in 5.4 [lex.pptoken] has a production reading

> each non-white-space character that cannot be one of the above

This is echoed in the max-munch rule in paragraph 3:

> If the input stream has been parsed into preprocessing tokens up to a given character, the next preprocessing token is the longest sequence of characters that could constitute a preprocessing token, even if that would cause further lexical analysis to fail.

This raises questions about the handling of raw string literals. Consider, for instance,

```
#define R "x"
const char* s = R"y";
```

The character sequence `R"y"` does not satisfy the syntactic requirements for a raw string. Should it be diagnosed as an ill-formed attempt at a raw string, or should it be well-formed, interpreting `R` as a preprocessor token that is a macro name and thus initializing `s` with a pointer to the string `"xy"`?

For another example, consider:

```
#define R "]"
const char* x = R"foo[";
```

Presumably this means that the entire rest of the file must be scanned for the characters `]foo"` and, if they are not found, macro-expand `R` and initialize `x` with a pointer to the string `"]foo["`. Is this the intended result?

Finally, does the requirement in 5.13.5 [lex.string] that

> A *d-char-sequence* shall consist of at most 16 characters.

mean that

```
#define R "x"
const char* y = R"12345678901234567[y]12345678901234567";
```

is ill-formed, or a valid initialization of `y` with a pointer to the string `"x12345678901234567[y]12345678901234567"`?

Additional note, June, 2009:

The translation of characters that are not in the basic source character set into universal-character-names in translation phase 1 raises an additional problem: each such character will occupy at least six of the 16 *r-char*s that are permitted. Thus, for example, `R"@@@[]@@@"` is ill-formed because `@@@` becomes `\u0040\u0040\u0040`, which is 18 characters.

One possibility for addressing this might be to disallow the `\` character completely as an *d-char*, which would have the effect of restricting *r-char*s to the basic source character set.

Proposed resolution (October, 2009):

1. Change the grammar in 5.13.5 [lex.string] as follows:

d-char:
any member of the basic source character set except:
space, the left square bracket `[`, the right square bracket `]`, **the backslash `\`,** and the control characters representing horizontal tab, vertical tab, form feed, and newline.

2. Change 5.13.5 [lex.string] paragraph 2 as follows:

A string literal that has an `R` in the prefix is a *raw string literal*. The *d-char-sequence* serves as a delimiter. The terminating *d-char-sequence* of a raw-string is the same sequence of characters as the initial *d-char-sequence*. A *d-char-sequence* shall consist of at most 16 characters. **If the input stream contains a sequence of characters that could be the prefix and initial double quote of a raw string literal, such as `R"`, those characters are considered to begin a raw string literal even if that literal is not well-formed. [*Example:***

```
#define R "x"
const char* s = R"y"; // ill-formed raw string, not "x" "y"
```

***—end example*]**

---

932. UCNs in closing delimiters of raw string literals

Section: 5.13.5 [lex.string] **Status:** CD2 **Submitter:** Alisdair Meredith **Date:** 7 July, 2009

[Voted into WP at March, 2010 meeting.]

Since members of the basic source character set can be written inside a string using a universal character name, it is not clear whether a UCN that represents `']'` or one of the characters in the terminating *d-char-sequence* should be interpreted as that character or as an attempt to "escape" that character and prevent its interpretation as part of the terminating sequence of a raw character string.

Notes from the July, 2009 meeting:

The CWG supported a resolution in which the *d-char-sequence* of a raw string literal is considered to be outside the literal and thus, by 5.3 [lex.charset] paragraph 2, could not contain a UCN designating a member of the basic source character set.

Proposed resolution (October, 2009):

Change 5.3 [lex.charset] paragraph 2 as follows:

Additionally, if the hexadecimal value for a universal-character-name outside **the *c-char-sequence*, *s-char-sequence*, or *r-char-sequence* of** a character or string literal corresponds to a control character (in either of the ranges 0x00-0x1F or 0x7F-0x9F, both inclusive) or to a character in the basic source character set, the program is ill-formed.

---

931. Confusing reference to the length of a user-defined string literal

Section: 5.13.8 [lex.ext] **Status:** CD2 **Submitter:** Alisdair Meredith **Date:** 6 July, 2009

[Voted into WP at March, 2010 meeting.]

5.13.8 [lex.ext] paragraph 5 says,

If *L* is a *user-defined-string-literal*, let *str* be the literal without its *ud-suffix* and let *len* be the number of characters (or code points) in *str* (i.e., its length excluding the terminating null character).

The length of a null-terminated string is defined in 20.4.2.1.5.1 [byte.strings] as the number of bytes preceding the terminator, but a single code point in a UTF-8 string can require more than one byte, so this sentence is inconsistent and needs to be revised to make clear which definition is in view.

Proposed resolution (October, 2009):

Change 5.13.8 [lex.ext] paragraph 5 as follows:

If *L* is a *user-defined-string-literal*, let *str* be the literal without its *ud-suffix* and let *len* be the number of ~~characters (or code points)~~ **code units** in *str* (i.e., its length excluding the terminating null character)...

---

633. Specifications for variables that should also apply to references

Section: 6 [basic] **Status:** CD2 **Submitter:** Alisdair Meredith **Date:** 17 May 2007

N2800 comment UK 22

[Voted into WP at March, 2010 meeting as document N2993.]

There are a number of specifications in the Standard that should also apply to references. For example:

- 6 [basic] paragraphs 3-4 indicate that a reference cannot have a name because it is not an entity. (See also issue 485.)
- 6.4.1 [basic.lookup.unqual] paragraph 13 covers unqualified lookup in the initializer of a variable member of a namespace but not that of a reference member of a namespace. It would be very strange if the lookup in these two cases were different.
- 6.5 [basic.link] paragraph 8 prohibits use of a type without linkage as the type of a variable with linkage, but not as the type of a reference with linkage. (References with linkage are explicitly mentioned earlier in the section.)
- 6.7.1 [basic.stc.static] paragraph 3 permits local static variables but not local static references.

A number of other examples could be cited. A thorough review is needed to make sure that references are completely specified.

Notes from the September, 2008 meeting:

The CWG expressed interest in an approach that would define "variable" to include both objects and references and to use that for both this issue and issue 570.

Proposed resolution (October, 2009):

See paper PL22.16/09-0183 = WG21 N2993. This resolution also resolves issue 570.

---

719. Specifications for *operator-function-id* that should also apply to *literal-operator-id*

Section: 6 [basic] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 19 September, 2008

[Voted into WP at October, 2009 meeting.]

When user-defined literals were added, a new form of operator function was created. Presumably many of the existing specifications that deal with *operator-function-id*s (the definition of *name*, for instance, in paragraph 4 of 6 [basic]) should also apply to *literal-operator-id*s.

Proposed resolution (June, 2009):

1. Change 6 [basic] paragraph 4 as follows:

   A *name* is a use of an *identifier* (5.10 [lex.name]), *operator-function-id* (16.5 [over.oper]), ***literal-operator-id*** **(16.5.8 [over.literal]),** *conversion-function-id* (15.3.2 [class.conv.fct]), or *template-id* (17.2 [temp.names]) that denotes an entity or *label* (9.6.4 [stmt.goto], 9.1 [stmt.label]).

2. Change _N4567_.5.1.1 [expr.prim.general] paragraph 3 as follows:

   The operator `::` followed by an *identifier*, a *qualified-id*, ~~or~~ an *operator-function-id***, or a *literal-operator-id*** is a *primary-expression*. Its type is specified by the declaration of the identifier, *qualified-id*, ~~or~~ *operator-function-id***, or *literal-operator-id***. The result is the entity denoted by the identifier, *qualified-id*, ~~or~~ *operator-function-id***, or *literal-operator-id***. The result is an lvalue if the entity is a function or variable. The identifier, *qualified-id*, ~~or~~ *operator-function-id***, or *literal-operator-id*** shall have global namespace scope or be visible in global scope because of a *using-directive* (10.3.4 [namespace.udir])...

3. Add the following production to the grammar for *qualified-id* in _N4567_.5.1.1 [expr.prim.general] paragraph 7:

   qualified-id:
   `::`$_{opt}$ *nested-name-specifier* `template`$_{opt}$ *unqualified-id*
   `::` *identifier*
   `::` *operator-function-id*
   `::` ***literal-operator-id***
   `::` *template-id*

4. Add the following production to the grammar for *template-id* in 17.2 [temp.names] paragraph 1:

   template-id:
   simple-template-id

operator-function-id < *template-argument-list*$_{opt}$ >
literal-operator-id < ***template-argument-list*$_{opt}$** >

5. Change 17.2 [temp.names] paragraph 3 as follows:

   After name lookup (6.4 [basic.lookup]) finds that a name is a *template-name*, or that an *operator-function-id* **or a *literal-operator-id*** refers to a set of overloaded functions any member of which is a function template...

6. Change 17.5 [temp.type] paragraph 1 bullet 1 as follows:

   - their *template-name*__s__**,** ~~or~~ *operator-function-id*__s__**, or *literal-operator-id*__s__** refer to the same template, and

---

942. Is `this` an entity?

Section: 6 [basic] **Status:** CD2 **Submitter:** Maurer **Date:** 14 July, 2009

[Voted into WP at March, 2010 meeting.]

At least in the new wording for 8.1.5 [expr.prim.lambda] paragraph 10 as found in paper N2927, `this` is explicitly assumed to be an entity. It should be investigated whether `this` should be added to the list of entities found in 6 [basic] paragraph 3.

Proposed resolution (October, 2009):

1. Change 6 [basic] paragraph 3 as follows:

   An *entity* is a value, object, variable, reference, function, enumerator, type, class member, template, template specialization, namespace, ~~or~~ parameter pack**, or** `this`.

2. Change 6.2 [basic.def.odr] paragraph 2 as follows:

   ...is immediately applied. `this` **is used if it appears as a potentially-evaluated expression (including as the result of the implicit transformation in the body of a non-static member function (12.2.2 [class.mfct.non-static])).** A virtual member function...

3. Delete 8.1.5 [expr.prim.lambda] paragraph 7:

   ~~For the purpose of describing the behavior of *lambda-expressions* below, `this` is considered to be "used" if replacing `this` by an invented variable `v` with automatic storage duration and the same type as `this` would result in `v` being used (6.2 [basic.def.odr]).~~

---

570. Are references subject to the ODR?

Section: 6.2 [basic.def.odr] **Status:** CD2 **Submitter:** Dave Abrahams **Date:** 2 April 2006

N2800 comment UK 26

[Voted into WP at March, 2010 meeting as document N2993.]

6.2 [basic.def.odr] paragraph 1 says,

> No translation unit shall contain more than one definition of any variable, function, class type, enumeration type or template.

This says nothing about references. Is it permitted to define a reference more than once in a single translation unit? (The list in paragraph 5 of things that can have definitions in multiple translation units does not include references.)

Notes from the September, 2008 meeting:

The CWG expressed interest in an approach that would define "variable" to include both objects and references and to use that for both this issue and issue 633.

Proposed resolution (October, 2009):

This issue is resolved by the resolution of issue 633.

---

481. Scope of template parameters

Section: 6.3 [basic.scope] **Status:** CD2 **Submitter:** Gabriel Dos Reis **Date:** 01 Nov 2004

N2800 comment FR 16

[Voted into WP at October, 2009 meeting.]

Sections 6.3.3 [basic.scope.block] to 6.3.7 [basic.scope.class] define and summarize different kinds of scopes in a C++ program. However it is missing a description for the scope of template parameters. I believe a section is needed there — even though some information may be found in clause 14.

Proposed resolution (September, 2009):

1. Insert the following as a new paragraph following 6.3.2 [basic.scope.pdecl] paragraph 8:

   The point of declaration of a template parameter is immediately after its complete *template-parameter*. [*Example:*

   ```
   typedef unsigned char T;
   template<class T
        = T          // Lookup finds the typedef name of unsigned char.

        , T          //Lookup finds the template parameter.
          N = 0> struct A {};
   ```

   —*end example*]

2. Delete 17.1 [temp.param] paragraph 14:

   ~~A *template-parameter* shall not be used in its own default argument.~~

   [Drafting note: This change conflicts with the resolution for issue 187 but is in accord with widespread implementation practice.]

3. Insert the following as a new section following 6.3.8 [basic.scope.enum]:

   Template Parameter Scope [basic.scope.temp]

   The declarative region of the name of a template parameter of a template *template-parameter* is the smallest *template-parameter-list* in which the name was introduced.

   The declarative region of the name of a template parameter of a template is the smallest *template-declaration* in which the name was introduced. Only template parameter names belong to this declarative region; any other kind of name introduced by the *declaration* of a *template-declaration* is instead introduced into the same declarative region where it would be introduced as a result of a non-template declaration of the same name. [*Example:*

   ```
   namespace N {
     template<class T> struct A{};              // line 2
     template<class U> void f(U){}              // line 3
     struct B {
       template<class V>friend int g(struct C*); // line 5
     };
   }
   ```

   The declarative regions of `T`, `U` and `V` are the *template-declaration*s on lines 2, 3 and 5, respectively. But the names `A`, `f`, `g` and `C` all belong to the same declarative region—namely, the *namespace-body* of `N`. (`g` is still considered to belong to this declarative region in spite of its being hidden during qualified and unqualified name lookup.) —*end example*]

   The potential scope of a template parameter name begins at its point of declaration (6.3.2 [basic.scope.pdecl]) and ends at the end of its declarative region. [*Note:* this implies that a *template-parameter* can be used in the declaration of subsequent *template-parameter*s and their default arguments but cannot be used in preceding *template-parameter*s or their default arguments. For example,

   ```
   template<class T, T* p, class U = T> class X { /* ... */ };
   template<class T> void f(T* p = new T);
   ```

   This also implies that a *template-parameter* can be used in the specification of base classes. For example,

   ```
   template<class T> class X : public Array<T> { /* ... */ };
   template<class T> class Y : public T { /* ... */ };
   ```

   The use of a template parameter as a base class implies that a class used as a template argument must be defined and not just declared when the class template is instantiated. —*end note*]

   The declarative region of the name of a template parameter is nested within the immediately-enclosing declarative region. [*Note:* as a result, a *template-parameter* hides any entity with the same name in an enclosing scope (6.3.10 [basic.scope.hiding]). [*Example:*

   ```
   typedef int N;
   template<N X, typename N, template<N Y> class T>
     struct A;
   ```

   Here, `X` is a non-type template parameter of type `int` and `Y` is a non-type template parameter of the same type as the second template parameter of `A`. —*end example*] —*end note*]

   [*Note:* because the name of a template parameter cannot be redeclared within its potential scope (17.7.1 [temp.local]), a template parameter's scope is often its potential scope. However, it is still possible for a template parameter name to be hidden; see 17.7.1 [temp.local]. —*end note*]

4. Delete 17.1 [temp.param] paragraph 13, including the example:

   ~~The scope of a *template-parameter* extends...~~

5. Delete 17.7.1 [temp.local] paragraph 6, including the note and example:

   ~~The scope of a *template-parameter* extends...~~

---

642. Definition and use of "block scope" and "local scope"

Section: 6.3.3 [basic.scope.block] **Status:** CD2 **Submitter:** Alisdair Meredith **Date:** 6 Aug 2007

[Voted into WP at March, 2010 meeting.]

The Standard uses the terms "block scope" and "local scope" interchangeably, but the former is never formally defined. Would it be better to use only one term consistently? "Block scope" seems to be more frequently used.

Notes from the October, 2007 meeting:

The CWG expressed a preference for the term "local scope."

Notes from the September, 2008 meeting:

Reevaluating the relative prevalence of the two terms (including the fact that new uses of "block scope" are being introduced, e.g., in both the lambda and thread-local wording) led to CWG reversing its previous preference for "local scope." The resolution will need to add a definition of "block scope" and should change the title of 6.3.3 [basic.scope.block].

Proposed resolution (October, 2009):

1. Change 6.3.2 [basic.scope.pdecl] paragraph 2 as follows:

   [*Note:* a ~~nonlocal~~ name **from an outer scope** remains visible up to the point of declaration of the ~~local~~ name that hides it. [*Example:*

   ```
   const int i = 2;
   { int i[i]; }
   ```

   declares a ~~local~~ **block-scope** array of two integers. —*end example*] —*end note*]

2. Change the section heading of 6.3.3 [basic.scope.block] from "Local scope" to "Block scope."

3. Change 6.3.3 [basic.scope.block] paragraph 1 as follows:

   A name declared in a block (9.3 [stmt.block]) is local to that block**; it has *block scope***. Its potential scope begins at its point of declaration (6.3.2 [basic.scope.pdecl]) and ends at the end of its ~~declarative region~~ **block. A variable declared at block scope is a *local variable***.

4. Change 6.3.3 [basic.scope.block] paragraph 3 as follows:

   The name ~~in a `catch` exception-declaration~~ **declared in an *exception-declaration*** is local to the ~~handler~~ ***handler*** and shall not be redeclared in the outermost block of the ~~handler~~ ***handler***.

5. Change 6.3.10 [basic.scope.hiding] paragraph 3 as follows:

   In a member function definition, the declaration of a ~~local~~ name **at block scope** hides the declaration of a member of the class with the same name...

6. Change 6.5 [basic.link] paragraph 8 as follows:

   ...Moreover, except as noted, a name declared ~~in a local~~ **at block** scope (6.3.3 [basic.scope.block]) has no linkage...

7. Change 6.6.3 [basic.start.dynamic] paragraph 1 as follows:

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

8. Change 6.6.3 [basic.start.dynamic] paragraph 2 as follows:

   If a function contains a ~~local~~ **block-scope** object of static or thread storage duration that has been destroyed and the function is called during the destruction of an object with static or thread storage duration, the program has undefined behavior if the flow of control passes through the definition of the previously destroyed ~~local~~ **block-scope** object. Likewise, the behavior is undefined if the ~~function-local~~ **block-scope** object is used indirectly (i.e., through a pointer) after its destruction.

9. Change 6.6.3 [basic.start.dynamic] paragraph 3 as follows:

   If the completion of the initialization of a ~~non-local~~ **non-block-scope** object with static storage duration is sequenced before a call to `std::atexit` (see `<cstdlib>`, 21.5 [support.start.term]), the call to the function passed to `std::atexit` is sequenced before the call to the destructor for the object. If a call to `std::atexit` is sequenced before the completion of the initialization of a ~~non-local~~ **non-block-scope** object with static storage duration, the call to the destructor...

[Editorial note: the occurrences of "non-local" in this change are removed by the proposed resolution for issue 946.]

10. Change 9.3 [stmt.block] paragraph 1 as follows:

    ...A compound statement defines a ~~local~~ **block** scope (6.3 [basic.scope])...

11. Change 9.4 [stmt.select] paragraph 1 as follows:

    ...The substatement in a *selection-statement* (each substatement, in the `else` form of the `if` statement) implicitly defines a ~~local~~ **block** scope (6.3 [basic.scope])...

12. Change 9.4 [stmt.select] paragraph 5 as follows:

    If a condition can be syntactically resolved as either an expression or the declaration of a ~~local~~ **block-scope** name, it is interpreted as a declaration.

13. Change 9.5 [stmt.iter] paragraph 2 as follows:

    The substatement in an *iteration-statement* implicitly defines a ~~local~~ **block** scope (6.3 [basic.scope]) which is entered and exited each time through the loop.

14. Change 9.7 [stmt.dcl] paragraph 3 as follows:

    ...A program that jumps[84] from a point where a ~~local~~ variable with automatic storage duration...

15. Change 9.7 [stmt.dcl] paragraph 4 as follows:

    The zero-initialization (11.6 [dcl.init]) of all ~~local~~ **block-scope** objects with static storage duration (6.7.1 [basic.stc.static]) or thread storage duration (6.7.2 [basic.stc.thread]) is performed before any other initialization takes place. Constant initialization (6.6.2 [basic.start.static]) of a ~~local~~ **block-scope** entity with static storage duration, if applicable, is performed before its block is first entered. An implementation is permitted to perform early initialization of other ~~local~~ **block-scope** objects...

16. Change 9.7 [stmt.dcl] paragraph 5 as follows:

    The destructor for a ~~local~~ **block-scope** object with static or thread storage duration will be executed if and only if the variable was constructed. [*Note:* 6.6.3 [basic.start.dynamic] describes the order in which ~~local~~ **block-scope** objects with static and thread storage duration are destroyed. *—end note*]

17. Change 11.4 [dcl.fct.def] paragraph 7 as follows:

    In the *function-body*, a *function-local predefined variable* denotes a ~~local~~ **block-scope** object of static storage duration that is implicitly defined (see 6.3.3 [basic.scope.block]).

18. Change the example in 12.1 [class.name] paragraph 2 as follows:

```
...
void g() {
  struct s;       // hide global struct s
                  // with a ~~local~~ block-scope declaration
...
```

19. Change the example in 12.1 [class.name] paragraph 3 as follows:

```
...
void g(int s) {
  struct s* p = new struct s;   // global s
  p->a = s;                     // ~~local~~ parameter s
}
```

---

490. Name lookup in friend declarations

Section: 6.4.1 [basic.lookup.unqual] **Status:** CD2 **Submitter:** Ben Hutchings **Date:** 7 Dec 2004

[Voted into WP at March, 2010 meeting.]

When 6.4.1 [basic.lookup.unqual] paragraph 10 says,

> In a `friend` declaration naming a member function, a name used in the function declarator and not part of a *template-argument* in a *template-id* is first looked up in the scope of the member function's class. If it is not found, or if the name is part of a *template-argument* in a *template-id*, the look up is as described for unqualified names in the definition of the class granting friendship.

what does "in the scope of the member function's class" mean? Does it mean that only members of the class and its base classes are considered? Or does it mean that the same lookup is to be performed as if the name appeared in the member function's class? Implementations vary in this regard. For example:

```
struct s1;

namespace ns {
    struct s1;
}
```

```
struct s2 {
    void f(s1 &);
};

namespace ns {
    struct s3 {
        friend void s2::f(s1 &);
    };
}
```

Microsoft Visual C++ and Comeau C++ resolve `s1` in the friend declaration to `ns::s1` and issue an error, while g++ resolves it to `::s1` and accepts the code.

Notes from the April, 2005 meeting:

The phrase "looked up in the scope of [a] class" occurs frequently throughout the Standard and always refers to the member name lookup described in 13.2 [class.member.lookup]. This is the first interpretation mentioned above ("only members of the class and its base classes"), resolving `s1` to `ns::s1`. A cross-reference to 13.2 [class.member.lookup] will be added to 6.4.1 [basic.lookup.unqual] paragraph 10 to make this clearer.

In discussing this question, the CWG noticed another problem: the text quoted above applies to all *template-argument*s appearing in the function declarator. The intention of this rule, however, is that only *template-argument*s in the *declarator-id* should ignore the member function's class scope; *template-argument*s used elsewhere in the function declarator should be treated like other names. For example:

```
template<typename T> struct S;
struct A {
  typedef int T;
  void foo(S<T>);
};
template <typename T> struct B {
  friend void A::foo(S<T>);  // i.e., S<A::T>
};
```

Proposed resolution (February, 2010):

Change 6.4.1 [basic.lookup.unqual] paragraph 10 as follows:

> In a `friend` declaration naming a member function, a name used in the function declarator and not part of a *template-argument* in ~~*a template-id*~~ **the *declarator-id*** is first looked up in the scope of the member function's class **(13.2 [class.member.lookup])**. If it is not found, or if the name is part of a *template-argument* in ~~*a template-id*~~ **the *declarator-id***, the look up is as described for unqualified names in the definition of the class granting friendship. [*Example:*
>
> ```
> struct A {
>   typedef int AT;
>   void f1(AT);
>   void f2(float);
>    template<typename T> void f3();
> };
> struct B {
>    typedef char AT;
>   typedef float BT;
>   friend void A::f1(AT);     // parameter type is A::AT
>   friend void A::f2(BT);     // parameter type is B::BT
>    friend void A::f3<AT>();   // template argument is B::AT
> };
> ```
>
> —*end example*]

---

598. Associated namespaces of overloaded functions and function templates

Section: 6.4.2 [basic.lookup.argdep] **Status:** CD2 **Submitter:** Mike Miller **Date:** 27 September 2006

[Voted into the WP at the March, 2009 meeting.]

The resolution of issue 33 added the following wording in 6.4.2 [basic.lookup.argdep]:

> In addition, if the argument is the name or address of a set of overloaded functions and/or function templates, its associated classes and namespaces are the union of those associated with each of the members of the set: the namespace in which the function or function template is defined and the classes and namespaces associated with its (non-dependent) parameter types and return type.

This wording is self-contradictory: although it claims that the treatment of overload sets is intended to be "the union of those associated with each of the members of the set," it says that the namespace of which each function or function template is a member is to be considered an associated namespace. That is different from the case of a non-overloaded function argument; in that case, because only the *type* of the argument is considered, the namespace of which the function is a member is *not* an associated namespace. This should be rectified so that overloaded and unoverloaded functions really are treated the same.

Proposed resolution (June, 2008):

Change 6.4.2 [basic.lookup.argdep] paragraph 2 as follows:

...In addition, if the argument is the name or address of a set of overloaded functions and/or function templates, its associated classes and namespaces are the union of those associated with each of the members of the set~~: the namespace in which the function or function template is defined and~~**, i.e.,** the classes and namespaces associated with its (non-dependent) parameter types and return type.

---

705. Suppressing argument-dependent lookup via parentheses

Section: 6.4.2 [basic.lookup.argdep] **Status:** CD2 **Submitter:** Mike Miller **Date:** 29 July, 2008

[Voted into WP at October, 2009 meeting.]

During the discussion of issue 704, some people expressed a desire to reconsider whether parentheses around the name of the function in a function call should suppress argument-dependent lookup, on the basis that this is overly subtle and not obvious. Others pointed out that this technique is used (both intentionally and inadvertently) in existing code and changing the behavior could cause problems.

It was also observed that the normative text that specifies this behavior is itself subtle, relying an a very precise interpretation of the preposition used in 6.4.2 [basic.lookup.argdep] paragraph 1:

> When an unqualified name is used as the *postfix-expression* in a function call...

This is taken to mean that something like `(f)(x)` is not subject to argument-dependent lookup because the name `f` is used *in* but not *as* the *postfix-expression*. This could be confusing, especially in light of the use of the term *postfix-expression* to refer to the name inside the parentheses, not to the parenthesized expression, in 16.3.1.1 [over.match.call] paragraph 1. If the decision is to preserve this effect of a parenthesized name in a function call, the wording should probably be revised to specify it more explicitly.

Notes from the September, 2008 meeting:

The CWG agreed that the suppression of argument-dependent lookup by parentheses surrounding the *postfix-expression* is widely known and used in the C++ community and must be preserved. The wording should be changed to make this effect clearer.

Proposed resolution (September, 2008):

Change 6.4.2 [basic.lookup.argdep] paragraph 1 as follows:

> When ~~an unqualified name is used as~~ the *postfix-expression* in a function call (8.2.2 [expr.call]) **is an *unqualified-id***, other namespaces not considered during the usual unqualified lookup (6.4.1 [basic.lookup.unqual]) may be searched...

Proposed resolution (September, 2009):

Change 6.4.2 [basic.lookup.argdep] paragraph 1 as follows:

> When ~~an unqualified name is used as~~ the *postfix-expression* in a function call (8.2.2 [expr.call]) **is an *unqualified-id***, other namespaces not considered during the usual unqualified lookup (6.4.1 [basic.lookup.unqual]) may be searched, and in those namespaces, namespace- scope friend function declarations (14.3 [class.friend]) not otherwise visible may be found. These modifications to the search depend on the types of the arguments (and for template template arguments, the namespace of the template argument). **[*Example:***
>
> ```
> namespace N {
>   struct S { };
>   void f(S);
> }
>
> void g() {
>   N::S s;
>   f(s);      // calls N::f
>   (f)(s);    // error: N::f not considered; parentheses prevent argument-dependent lookup
> }
> ```
>
> ***—end example*]**

1000. Mistaking member typedefs for constructors

Section: 6.4.3.1 [class.qual] **Status:** CD2 **Submitter:** Jason Merrill **Date:** 20 November, 2009

[Voted into WP at March, 2010 meeting.]

The recent addition to support inherited constructors changed 6.4.3.1 [class.qual] paragraph 2 to say that

> if the name specified after the *nested-name-specifier* is the same as the *identifier* or the *simple-template-id*'s *template-name* in the last component of the *nested-name-specifier*,

the *qualified-id* is considered to name a constructor. This causes problems for a common naming scheme used in some class libraries:

```
struct A {
  typedef int type;
};
```

```
struct B {
  typedef A type;
};
B::type::type t;
```

This change causes this to name the `A` constructor instead of the `A::type` typedef.

Proposed resolution (February, 2010):

Change 6.4.3.1 [class.qual] paragraph 2 as follows:

> In a lookup in which the constructor is an acceptable lookup result and the *nested-name-specifier* nominates a class `C`:
>
> - if the name specified after the *nested-name-specifier*, when looked up in `C`, is the injected-class-name of `C` (Clause 12 [class]), or
> - **in a *using-declaration* (10.3.3 [namespace.udecl]) that is a *member-declaration*,** if the name specified after the *nested-name-specifier* is the same as the *identifier* or the *simple-template-id*'s *template-name* in the last component of the *nested-name-specifier*,
>
> the name is instead considered to name the constructor of class `C`...

---

861. Unintended ambiguity in inline namespace lookup

Section: 6.4.3.2 [namespace.qual] **Status:** CD2 **Submitter:** Michael Wong **Date:** 7 April, 2009

[Voted into WP at March, 2010 meeting as part of document N3079.]

The algorithm for namespace-qualified lookup is given in 6.4.3.2 [namespace.qual] paragraph 2:

> Given `X::m` (where `X` is a user-declared namespace), or given `::m` (where `X` is the global namespace), let `S` be the set of all declarations of `m` in `X` and in the transitive closure of all namespaces nominated by *using-directive*s in `X` and its used namespaces, except that *using-directive*s that nominate non-inline namespaces (10.3.1 [namespace.def]) are ignored in any namespace, including `X`, directly containing one or more declarations of `m`.

Consider the following example:

```
namespace A {
    inline namespace B {
        namespace C {
            int i;
        }
        using namespace C;
    }
    int i;
}

int j = A::i;     // ambiguous
```

The transitive closure includes `B` because it is `inline`, and it includes `C` because there is no declaration of `i` in `B`. As a result, `A::i` finds both the `i` declared in `A` and the one declared in `C`, and the lookup is ambiguous.

This result is apparently unintended.

Proposed resolution (November, 2009):

1. Change 10.3.1 [namespace.def] paragraph 9 as follows:

   > These properties are transitive: if a namespace `N` contains an inline namespace `M`, which in turn contains an inline namespace `O`, then the members of `O` can be used as though they were members of `M` or `N`. **The transitive closure of all inline namespaces in `N` is the *inline namespace set* of `N`.** The set of namespaces consisting of the innermost non-inline namespace enclosing an inline namespace `O`, together with any intervening inline namespaces, is the *enclosing namespace set* of `O`.

2. Insert a new paragraph before 6.4.3.2 [namespace.qual] paragraph 2 and change the existing paragraph 2 as follows:

   > **For a namespace `X` and name `m`, the namespace-qualified lookup set `S(X,m)` is defined as follows: Let `S'(X,m)` be the set of all declarations of `m` in `X` and the inline namespace set of `X` (10.3.1 [namespace.def]). If `S'(X,m)` is not empty, `S(X,m)` is `S'(X,m)`; otherwise, `S(X,m)` is the union of `S(N`$_i$`,m)` for all non-inline namespaces `N`$_i$ nominated by *using-directive*s in `X` and its inline namespace set.**
   >
> Given `X::m` (where `X` is a user-declared namespace), or given `::m` (where `X` is the global namespace), ~~let `S` be the set of all declarations of `m` in `X` and in the transitive closure of all namespaces nominated by *using-directive*s in `X` and its used namespaces, except that *using-directive*s that nominate non-inline namespaces (10.3.1 [namespace.def]) are ignored in any namespace, including `X`, directly containing one or more declarations of `m`. No namespace is searched more than once in the lookup of a name. If~~ **if** `S(X,m)` is the empty set, the program is ill-formed. Otherwise, if `S(X,m)` has exactly one member, or if the context of the reference is a *using-declaration* (10.3.3 [namespace.udecl]), `S(X,m)` is the required set of declarations of `m`. Otherwise if the use of `m` is not one that allows a unique declaration to be chosen from `S(X,m)`, the program is ill-formed. [*Example*:...

---

527. Problems with linkage of types

Section: 6.5 [basic.link] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 28 July 2005

[Voted into WP at October, 2009 meeting.]

The resolution of issue 389 makes code like

```
static struct {
    int i;
    int j;
} X;
```

ill-formed. This breaks a lot of code for no apparent reason, since the name `X` is not known outside the translation unit in which it appears; there is therefore no danger of collision and no need to mangle its name.

There has also been recent discussion on the email reflectors as to whether the restrictions preventing use of types without linkage as template arguments is needed or not, with the suggestion that a mechanism like that used to give members of the unnamed namespace unique names could be used for unnamed and local types. See also issue 488, which would become moot if types without linkage could be used as template parameters.

Notes from the October, 2005 meeting:

The Evolution Working Group is discussing changes that would address this issue. CWG will defer consideration until the outcome of the EWG discussions is clear.

Notes from the April, 2006 meeting:

The CWG agreed that the restriction in 6.5 [basic.link] paragraph 8 on use of a type without linkage should apply only to variables and functions with external linkage, not to variables and functions with internal linkage (i.e., the example should be accepted). This is a separate issue from the question before the EWG and should be resolved independently.

Additional note (April, 2006):

Even the restriction of the rule to functions and objects with external linkage may not be exactly what we want. Consider an example like:

```
namespace {
    struct { int i; } s;
}
```

The variable `s` has external linkage but can't be named outside its translation unit, so there's again no reason to prohibit use of a type without linkage in its declaration.

Notes from the June, 2008 meeting:

Paper N2657, adopted at the June, 2008 meeting, allows local and unnamed types to be used as template parameters. That resolution is narrowly focused, however, and does not address this issue.

Proposed resolution (June, 2009):

Change 6.5 [basic.link] paragraph 8 as follows:

> ...A type without linkage shall not be used as the type of a variable or function with **external** linkage, unless
>
> - the variable or function has ~~extern "C"~~ **C language** linkage (10.5 [dcl.link]), or
> - **the variable or function is declared within an unnamed namespace (10.3.1 [namespace.def]), or**
> - the variable or function is not used (6.2 [basic.def.odr]) or is defined in the same translation unit.

[Drafting note: the context shown for the preceding resolution assumes that the resolution for issue 757 has been applied.]

---

571. References declared `const`

Section: 6.5 [basic.link] **Status:** CD2 **Submitter:** Dave Abrahams **Date:** 31 March 2006

[Voted into the WP at the March, 2009 meeting.]

According to 6.5 [basic.link] paragraph 3,

> A name having namespace scope (6.3.6 [basic.scope.namespace]) has internal linkage if it is the name of
>
> - an object, reference, function or function template that is explicitly declared `static` or,
> - an object or reference that is explicitly declared `const` and neither explicitly declared `extern` nor previously declared to have external linkage;

It is not possible to declare a reference to be `const`.

Proposed resolution (March, 2008):

Change 6.5 [basic.link] paragraph 3 as indicated (note addition of punctuation in the first bullet):

> A name having namespace scope (6.3.6 [basic.scope.namespace]) has internal linkage if it is the name of
>
> - an object, reference, function**,** or function template that is explicitly declared `static`**;** or,
> - an object ~~or reference~~ that is explicitly declared `const` and neither explicitly declared `extern` nor previously declared to have external linkage; or
> - a data member of an anonymous union.

---

757. Types without linkage in declarations

Section: 6.5 [basic.link] **Status:** CD2 **Submitter:** John Spicer **Date:** 23 December, 2008

[Voted into WP at July, 2009 meeting.]

Paper N2657, adopted at the June, 2008 meeting, removed the prohibition of local and unnamed types as template arguments. As part of the change, 6.5 [basic.link] paragraph 8 was modified to read,

> A type without linkage shall not be used as the type of a variable or function with linkage, unless
>
> - the variable or function has extern "C" linkage (10.5 [dcl.link]), or
> - the type without linkage was named using a dependent type (17.7.2.1 [temp.dep.type]).

Because a type without linkage can only be named as a dependent type, there are still some potentially useful things that cannot be done:

```
template <class T> struct A {
  friend void g(A, T);  // this can't be defined later
  void h(T);  // this cannot be explicitly specialized
};

template <class T> void f(T) {
  A<T> at;
  g(at, (T)0);
}

enum { e };

void g(A<decltype(e)>, decltype(e)){}  // not allowed

int main() {
  f(e);
}
```

These deficiencies could be addressed by allowing types without linkage to be used as the type of a variable or function, but with the requirement that any such entity that is used must also be defined in the same translation unit. This would allow issuing a compile-time, instead of a link-time, diagnostic if the definition were not provided, for example. It also seems to be easier to implement than the current rules.

Proposed resolution (March, 2009):

Change 6.5 [basic.link] paragraph 8 as follows:

> ...A type without linkage shall not be used as the type of a variable or function with linkage, unless
>
> - the variable or function has extern "C" linkage (10.5 [dcl.link]), or
> - ~~the type without linkage was named using a dependent type (17.7.2.1 [temp.dep.type])~~ **the variable or function is not used (6.2 [basic.def.odr]) or is defined in the same translation unit**.
>
> [*Note:* in other words, a type without linkage contains a class or enumeration that cannot be named outside its translation unit. An entity with external linkage declared using such a type could not correspond to any other entity in another translation unit of the program and thus ~~is not permitted~~ **must be defined in the translation unit if it is used**. Also note that classes with linkage may contain members whose types do not have linkage, and that typedef names are ignored in the determination of whether a type has linkage. —*end note*] [*Example:*
>
> ~~~~
> ~~void f() {~~
> ~~  struct A { int x; };    // no linkage~~
> ~~  extern A a;             // ill-formed~~
> ~~  typedef A B;~~
> ~~  extern B b;             // ill-formed~~
> ~~}~~
> ~~~~
>
> ~~—*end example*]~~
>
> ~~[*Example:*~~

~~template <class T> struct A {~~
~~// in A<X>, the following is allowed because the type with no linkage~~
~~// X is named using template parameter T.~~
~~friend void f(A, T) {}~~
~~};~~

~~template <class T> void g(T t) {~~
~~A<T> at;~~
~~f(at, t);~~
~~}~~

~~int main() {~~
~~class X {} x;~~
~~g(x);~~
~~}~~

```
template <typename T> struct B {
    void g(T){}
    void h(T);
    friend void i(B, T){}
};

void f() {
    struct A { int x; };  // no linkage
    A a = {1};
    B<A> ba;              // declares B<A>::g(A) and B<A>::h(A)
    ba.g(a);              // OK
    ba.h(a);              // error: B<A>::h(A) not defined in the translation unit
    i(ba, a);             // OK
}
```

—end example]

[Drafting note: issue 527 also changes part of the same text.]

---

966. Nested types without linkage

Section: 6.5 [basic.link] **Status:** CD2 **Submitter:** Jason Merrill **Date:** 15 September, 2009

[Voted into WP at March, 2010 meeting.]

The recent changes to allow use of unnamed types as template arguments require some rethinking of how unnamed types are treated in general. At least, a class-scope unnamed type should have the same linkage as its containing class. For example:

```
// File "hdr.h"
struct S {
  static enum { No, Yes } locked;
};
template<class T> void f(T);

// File "impl1.c"
#include "hdr.h"
template void f(decltype(S::locked));

// File "impl2.c"
#include "hdr.h"
template void f(decltype(S::locked));
```

The two explicit instantiation directives should refer to the same specialization.

Proposed resolution (February, 2010):

Change 6.5 [basic.link] paragraph 8 as follows:

> Names not covered by these rules have no linkage. Moreover, except as noted, a name declared in a local scope (6.3.3 [basic.scope.block]) has no linkage. A type is said to have linkage if and only if:
>
> - it is a class or enumeration type that is named (or has a name for linkage purposes (10.1.3 [dcl.typedef])) and the name has linkage; or
> - **it is an unnamed class or enumeration member of a class with linkage; or**
> - ...

---

792. Effects of `std::quick_exit`

Section: 6.6.1 [basic.start.main] **Status:** CD2 **Submitter:** US **Date:** 3 March, 2009

N2800 comment US 24

[Voted into WP at October, 2009 meeting.]

6.6.1 [basic.start.main] paragraph 4 discusses the effects of calling `std::exit` but says nothing about `std::quick_exit`.

Proposed resolution (July, 2009):

Change 6.6.1 [basic.start.main] paragraph 4 as follows:

~~Calling the function `std::exit(int)` declared in <cstdlib> (21.5 [support.start.term]) terminates~~ **Terminating** the program without leaving the current block **(e.g., by calling the function `std::exit(int)` (21.5 [support.start.term]))** ~~and hence without destroying~~ **does not destroy** any objects with automatic storage duration (15.4 [class.dtor])...

---

882. Defining `main` as deleted

Section: 6.6.1 [basic.start.main] **Status:** CD2 **Submitter:** Steve Adamczyk **Date:** 27 April, 2009

It should be stated in 6.6.1 [basic.start.main] that it a program that defines `main` as deleted is ill-formed.

Proposed resolution (July, 2009):

Change 6.6.1 [basic.start.main] paragraph 3 as follows:

> ...A program that declares `main` to be `inline`, `static`, or `constexpr`**, or that defines `main` as deleted,** is ill-formed...

---

776. Delegating constructors, destructors, and `std::exit`

Section: 6.6.3 [basic.start.dynamic] **Status:** CD2 **Submitter:** Michael Wong **Date:** 12 February, 2009

[Voted into WP at October, 2009 meeting.]

According to 6.6.3 [basic.start.dynamic] paragraph 1,

> Destructors (15.4 [class.dtor]) for initialized objects with static storage duration are called as a result of returning from `main` and as a result of calling `std::exit` (21.5 [support.start.term]).

It is unclear, in the presence of delegating constructors, exactly what an "initialized object" is. 6.8 [basic.life] paragraph 1 says that the lifetime of an object does not begin until it is completely initialized, i.e., when its principal constructor finishes execution. 18.2 [except.ctor] paragraph 2 says that an exception during the construction of class object only invokes destructors for fully-constructed base and member sub-objects (those for which the principal constructor has completed). On the other hand, the destructor for a complete class object is called if its non-delegating constructor has completed, even if the principal constructor has not yet finished. Which of these models is appropriate for the behavior of `std::exit`?

Notes from the March, 2009 meeting:

The CWG agreed that the destructor for a complete object should be called by `std::exit` if its non-delegating constructor has finished, just as for an exception.

Notes from the July, 2009 meeting:

The CWG decided that the direction adopted at the March, 2009 meeting was incorrect. Instead, the model should be the way completely-constructed base and member subobjects are handled: their destructors are called when an exception is thrown but not when `std::exit` is called.

Proposed resolution (July, 2009):

Change 6.6.3 [basic.start.dynamic] paragraph 1 as follows:

> Destructors (15.4 [class.dtor]) for initialized objects **(that is, objects whose lifetime (6.8 [basic.life]) has begun)** with static storage duration are called as a result of returning from main and as a result of calling `std::exit` (21.5 [support.start.term]). Destructors for initialized objects with thread storage duration...

---

946. Order of destruction of local static objects and calls to `std::atexit`

Section: 6.6.3 [basic.start.dynamic] **Status:** CD2 **Submitter:** Fraser Ross **Date:** 28 July, 2009

[Voted into WP at March, 2010 meeting.]

21.5 [support.start.term] paragraph 7 says that the order of destruction of objects with static storage duration and calls to functions registered by calling `std::atexit` is given in 6.6.3 [basic.start.dynamic]. Paragraph 1 of 6.6.3 [basic.start.dynamic] says,

> If the completion of the constructor or dynamic initialization of an object with static storage duration is sequenced before that of another, the completion of the destructor of the second is sequenced before the initiation of the destructor of the first.

This wording covers both local and namespace-scope objects, so it fixes the relative ordering of local object destructors with respect to those of namespace scope. Paragraph 3 says,

> If the completion of the initialization of a non-local object with static storage duration is sequenced before a call to `std::atexit` (see `<cstdlib>`, 21.5 [support.start.term]), the call to the function passed to `std::atexit` is sequenced before the call to the destructor for the object. If a call to `std::atexit` is sequenced before the completion of the initialization of a non-local object with static storage duration, the call to the destructor for the object is sequenced before the call to the function passed to `std::atexit`.

This fixes the relative ordering of destructors for namespace scope objects with respect to calls of `atexit` functions. However, the relative ordering of local destructors and `atexit` functions is left unspecified.

In the 2003 Standard, this was clear: 18.3 paragraph 8 said,

> A local static object `obj3` is destroyed at the same time it would be if a function calling the `obj3` destructor were registered with `atexit` at the completion of the `obj3` constructor.

Proposed resolution (October, 2009):

Change 6.6.3 [basic.start.dynamic] paragraph 3 as follows:

> If the completion of the initialization of ~~a non-local~~ **an** object with static storage duration is sequenced before a call to `std::atexit` (see `<cstdlib>`, 21.5 [support.start.term]), the call to the function passed to `std::atexit` is sequenced before the call to the destructor for the object. If a call to `std::atexit` is sequenced before the completion of the initialization of ~~a non-local~~ **an** object with static storage duration, the call to the destructor for the object is sequenced before the call to the function passed to `std::atexit`...

---

735. Missing case in specification of safely-derived pointers

Section: 6.7.4.3 [basic.stc.dynamic.safety] **Status:** CD2 **Submitter:** Jens Maurer **Date:** 14 October, 2008

N2800 comment DE 3

[Voted into WP at October, 2009 meeting.]

The bullets in 6.7.4.3 [basic.stc.dynamic.safety] paragraph 2 do not appear to cover the following example:

```
int& i = *new int(5);
// do something with i
delete &i;
```

Should `&i` be a safely-derived pointer value?

Proposed resolution (September, 2009):

Change 6.7.4.3 [basic.stc.dynamic.safety] paragraph 2, bullet 2, as follows:

- the result of taking the address of ~~a subobject of an lvalue~~ **an object (or one of its subobjects) designated by an lvalue** resulting from dereferencing a safely-derived pointer value;

---

853. Support for relaxed pointer safety

Section: 6.7.4.3 [basic.stc.dynamic.safety] **Status:** CD2 **Submitter:** Jens Maurer **Date:** 3 April, 2009

[Voted into WP at March, 2010 meeting.]

According to 23.10.4 [util.dynamic.safety] paragraph 16, when `std::get_pointer_safety()` returns `std::pointer_safety::relaxed`,

> pointers that are not safely derived will be treated the same as pointers that are safely derived for the duration of the program.

However, 6.7.4.3 [basic.stc.dynamic.safety] paragraph 4 says unconditionally that

> If a pointer value that is not a safely-derived pointer value is dereferenced or deallocated, and the referenced complete object is of dynamic storage duration and has not previously been declared reachable (23.10.4 [util.dynamic.safety]), the behavior is undefined.

This is a contradiction: the library clause attempts to constrain undefined behavior, which by definition is unconstrained.

Proposed resolution (July, 2009):

Change 6.7.4.3 [basic.stc.dynamic.safety] paragraph 4 as follows to define the terms “strict pointer safety” and “relaxed pointer safety,” which could then be used by the library clauses to achieve the desired effect:

An implementation may have *relaxed pointer safety*, in which case the validity of a pointer value does not depend on whether it is a safely-derived pointer value or not. Alternatively, an implementation may have *strict pointer safety*, in which case if ~~If~~ a pointer value that is not a safely-derived pointer value is dereferenced or deallocated, and the referenced complete object is of dynamic storage duration and has not previously been declared reachable (23.10.4 [util.dynamic.safety]), the behavior is undefined. [*Note:* this is true even if the unsafely-derived pointer value might compare equal to some safely-derived pointer value. —*end note*] **It is implementation-defined whether an implementation has relaxed or strict pointer safety.**

---

793. Use of class members during destruction

Section: 6.8 [basic.life] **Status:** CD2 **Submitter:** US **Date:** 3 March, 2009

N2800 comment US 26

[Voted into WP at March, 2010 meeting.]

Read literally, 6.8 [basic.life] paragraphs 1 and 5 would make any access to non-static members of a class from the class's destructor undefined behavior. This is clearly not the intent.

Proposed resolution (October, 2009):

Change 6.8 [basic.life] paragraphs 5-6 as follows:

> ...any pointer that refers to the storage location where the object will be or was located may be used but only in limited ways. ~~Such~~ **For an object under construction or destruction, see 15.7 [class.cdtor]. Otherwise, such** a pointer refers to allocated storage...
>
> ...any lvalue which refers to the original object may be used but only in limited ways. ~~Such~~ **For an object under construction or destruction, see 15.7 [class.cdtor]. Otherwise, such** an lvalue refers to allocated storage...

---

883. `std::memcpy` **vs** `std::memmove`

Section: 6.9 [basic.types] **Status:** CD2 **Submitter:** Lawrence Crowl **Date:** 29 April, 2009

[Voted into WP at October, 2009 meeting.]

The `std::memcpy` library function is singled out for special treatment in 6.9 [basic.types] paragraph 3:

> For any trivially copyable type `T`, if two pointers to `T` point to distinct `T` objects `obj1` and `obj2`, where neither `obj1` nor `obj2` is a base-class subobject, if the value of `obj1` is copied into `obj2`, using the `std::memcpy` library function, `obj2` shall subsequently hold the same value as `obj1`.

This specification should not be restricted to `std::memcpy` but should apply to any bytewise copying, including `std::memmove` (as is done in the footnote in the preceding paragraph, for example).

Proposed resolution (July, 2009):

Change 6.9 [basic.types] paragraph 3 as follows:

> For any trivially copyable type `T`, if two pointers to `T` point to distinct `T` objects `obj1` and `obj2`, where neither `obj1` nor `obj2` is a base-class subobject, if the ~~value of~~ **underlying bytes (4.4 [intro.memory]) making up** `obj1` ~~is~~ **are** copied into `obj2`~~, using the std::memcpy library function~~ **[*Footnote*: By using, for example, the library functions (20.5.1.2 [headers])** `std::memcpy` **or** `std::memmove`**. —*end footnote*]**, `obj2` shall subsequently hold the same value as `obj1`. [*Example:*...

---

846. Rvalue references to functions

Section: 6.10 [basic.lval] **Status:** CD2 **Submitter:** Daniel Krügler **Date:** 23 March, 2009

[Voted into WP at March, 2010 meeting as document N3055.]

The status of rvalue references to functions is not clear in the current wording. For example, 6.10 [basic.lval] paragraph 2 says,

> An lvalue refers to an object or function. Some rvalue expressions—those of (possibly cv-qualified) class or array type—also refer to objects. [*Footnote:* Expressions such as invocations of constructors and of functions that return a class type refer to objects, and the implementation can invoke a member function upon such objects, but the expressions are not lvalues. —*end footnote*]

This would tend to indicate that there are no rvalues of function type. However, 8 [expr] paragraph 6 says,

> If an expression initially has the type "rvalue reference to T" (11.3.2 [dcl.ref], 11.6.3 [dcl.init.ref]), the type is adjusted to "T" prior to any further analysis, and the expression designates the object or function denoted by the rvalue reference. If the expression is the result of calling a function, whether implicitly or explicitly, it is an rvalue; otherwise, it is an lvalue.

This explicitly indicates that rvalue references to functions are possible and that, in some cases, they yield function-typed rvalues. Furthermore, _N2914_.20.2.4 [concept.operator] paragraph 20 describes the concept `Callable` as:

```
auto concept Callable<typename F, typename... Args> {
  typename result_type;
  result_type operator()(F&, Args...);
  result_type operator()(F&&, Args...);
}
```

It would be strange if `Callable` were satisfied for a function object type but not for a function type.

However, assuming that rvalue references to functions are intended to be supported, it is not clear how an rvalue of function type is supposed to behave. For instance, 8.2.2 [expr.call] paragraph 1 says,

> For an ordinary function call, the postfix expression shall be either an lvalue that refers to a function (in which case the function-to-pointer standard conversion (7.3 [conv.func]) is suppressed on the postfix expression), or it shall have pointer to function type.

From this, it appears that an rvalue of function type cannot be used in a function call. It can't be converted to a pointer to function, either, as 7.3 [conv.func] paragraph 1 says,

> An lvalue of function type `T` can be converted to an rvalue of type "pointer to `T`." The result is a pointer to the function.

(See also issues 664 and especially 690. The approach described in the latter issue, viewing rvalue references as essentially lvalues rather than as essentially rvalues, could resolve the specification problems described above by eliminating the concept of an rvalue of function type.)

Proposed resolution (February, 2010):

See paper N3030.

---

693. New string types and deprecated conversion

Section: 7.2 [conv.array] **Status:** CD2 **Submitter:** Alisdair Meredith **Date:** 21 April, 2008

N2800 comment DE 4

[Voted into WP at October, 2009 meeting.]

The deprecated conversion from string literal to pointer to (non-const) character in 7.2 [conv.array] paragraph 2 has been extended to apply to `char16_t` and `char32_t` types, but not to UTF8 and raw string literals. Is this disparity intentional? Should it be extended to all new string types, reverted to just the original character types, or revoked altogether?

Additional places in the Standard that may need to change include 18.1 [except.throw] paragraph 3 and 16.3.3.2 [over.ics.rank] paragraph 3.

Additional discussion (August, 2008):

The removal of this conversion for current string literals would affect overload resolution for existing programs. For example,

```
struct S {
    S(const char*);
};
int f(char *);
int f(X);
int i = f("hello");
```

If the conversion were removed, the result would be a quiet change in behavior. Another alternative to consider would be a required diagnostic (without making the program ill-formed).

Notes from the September, 2008 meeting:

The CWG agreed that the deprecated conversion should continue to apply to the literals to which it applied in C++ 2003. Consensus was not reached regarding whether it should apply only to those literals or to all the new literals as well, although it was agreed that the current situation in which it applies to some, but not all, of the new literals is unacceptable.

Notes from the July, 2009 meeting:

The CWG reached consensus that the deprecated conversion should be removed altogether.

Proposed resolution (September, 2009):

1. Remove 7.2 [conv.array] paragraph 2:

> ~~A string literal (5.13.5 [lex.string]) with no prefix, with a `u` prefix, with a `U` prefix, or with an `L` prefix can be converted to an rvalue of type "pointer to `char`", "pointer to `char16_t`", "pointer to `char32_t`", or "pointer to `wchar_t`", respectively. In~~

~~any case, the result is a pointer to the first element of the array. This conversion is considered only when there is an explicit appropriate pointer target type, and not when there is a general need to convert from an lvalue to an rvalue. [*Note:* this conversion is deprecated. See Annex D [depr]. — *end note*] For the purpose of ranking in overload resolution (16.3.3.1.1 [over.ics.scs]), this conversion is considered an array-to-pointer conversion followed by a qualification conversion (7.5 [conv.qual]). [*Example:* "abc" is converted to "pointer to const char" as an array-to-pointer conversion, and then to "pointer to char" as a qualification conversion. — *end example*]~~

2. Delete the indicated text from the third sub-bullet of the first bullet of paragraph 3 of 16.3.3.2 [over.ics.rank]:

   - S1 and S2 differ only in their qualification conversion and yield similar types T1 and T2 (7.5 [conv.qual]), respectively, and the cv-qualification signature of type T1 is a proper subset of the cv-qualification signature of type T2~~, and S1 is not the deprecated string literal array-to-pointer conversion (4.2)~~. [*Example:* ...

3. Delete the note from 18.1 [except.throw] paragraph 3 as follows:

   A *throw-expression* initializes a temporary object, called the *exception object*, the type of which is determined by removing any top-level *cv-qualifier*s from the static type of the operand of `throw` and adjusting the type from "array of T" or "function returning T" to "pointer to T" or "pointer to function returning T", respectively. ~~[*Note:* the temporary object created for a *throw-expression* that is a string literal is never of type char*, char16_t*, char32_t*, or wchar_t*; that is, the special conversions for string literals from the types "array of const char", "array of const char16_t", "array of const char32_t", and "array of const wchar_t" to the types "pointer to char", "pointer to char16_t", "pointer to char32_t", and "pointer to wchar_t", respectively (7.2 [conv.array]), are never applied to a *throw-expression*. — *end note*]~~ The temporary is an lvalue...

4. Change the discussion of 5.13.5 [lex.string] in C.1.1 [diff.lex] as follows:

   Change: String literals made const
   The type of a string literal is changed... "array of `const wchar_t`."

   ```
   char* p = "abc";    // valid in C, invalid in C++
   ```

   ...

   Difficulty of converting: ~~Simple syntactic transformation, because string literals can be converted to char*; (7.2 [conv.array]). The most common cases are handled by a new but deprecated standard conversion~~ **Syntactic transformation. The fix is to add a cast**:

   ```
   ~~char* p = "abc";                  // valid in C, deprecated in C++~~
   ~~char* q = expr ? "abc" : "de";    // valid in C, invalid in C++~~
   void f(char*) {
       char* p = (char*)"abc";   // cast added
       f(p);
       f((char*)"def");          // cast added
   }
   ```

5. Delete _N3000_.D.4 [depr.string]:

   ~~**D.4 Implicit conversion from const strings [depr.string]**~~

   ~~The implicit conversion from const to non-const qualification for string literals (7.2 [conv.array]) is deprecated.~~

---

685. Integral promotion of enumeration ignores fixed underlying type

Section: 7.6 [conv.prom] **Status:** CD2 **Submitter:** Alberto Ganesh Barbati **Date:** 6 January, 2008

[Voted into WP at July, 2009 meeting.]

According to 7.6 [conv.prom] paragraph 2,

> An rvalue of an unscoped enumeration type (10.2 [dcl.enum]) can be converted to an rvalue of the first of the following types that can represent all the values of the enumeration (i.e. the values in the range b_{min} to b_{max} as described in 10.2 [dcl.enum]):
> `int`, `unsigned int`, `long int`, `unsigned long int`, `long long int`, or `unsigned long long int`.

This wording may have surprising behavior in this case:

```
enum E: long { e };

void f(int);
void f(long);

void g() {
    f(e);    // Which f is called?
}
```

Intuitively, as the programmer has explicitly expressed preference for `long` as the underlying type, he/she might expect `f(long)` to be called. However, if `long` and `int` happen to have the same size, then `e` is promoted to `int` (as it is the first type in the list that can represent all values of `E`) and `f(int)` is called instead.

According to 10.2 [dcl.enum] the underlying type of an enumeration is always well-defined for both the fixed and the non-fixed cases, so it makes sense simply to promote to the underlying type unless such a type would itself require promotion.

Suggested resolution:

In 7.6 [conv.prom] paragraph 2, replace all the text from "An rvalue of an unscoped enumeration type" through the end of the paragraph with the following:

> An rvalue of an unscoped enumeration type (10.2 [dcl.enum]) is converted to an rvalue of its underlying type if it is different from `char16_t`, `char32_t`, `wchar_t`, or has integer conversion rank greater than or equal to `int`. Otherwise, it is 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 int`, `unsigned long int`, `long long int`, or `unsigned long long int`.

(Note that this wording no longer needs to mention extended integer types as special cases.)

Proposed resolution (August, 2008):

Move the following text from 7.6 [conv.prom] paragraph 2 into a separate paragraph, making the indicated changes, and add the following new paragraph after it:

> An rvalue of an unscoped enumeration type **whose underlying type is not fixed** (10.2 [dcl.enum]) can be converted to an rvalue of the first of the following types that can represent all the values of the enumeration (i.e. the values in the range b_{min} to b_{max} as described in 10.2 [dcl.enum]): `int`, `unsigned int`, `long int`, `unsigned long int`, `long long int`, or `unsigned long long int`. If none of the types in that list can represent all the values of the enumeration, an rvalue of an unscoped enumeration type can be converted to an rvalue of the extended integer type with lowest integer conversion rank (7.14 [conv.bool]) greater than the rank of `long long` in which all the values of the enumeration can be represented. If there are two such extended types, the signed one is chosen.
>
> **An rvalue of an unscoped enumeration type whose underlying type is fixed (10.2 [dcl.enum]) can be converted to an rvalue of its underlying type. Moreover, if integral promotion can be applied to its underlying type, an rvalue of an unscoped enumeration type whose underlying type is fixed can also be converted to an rvalue of the promoted underlying type.**

---

707. Undefined behavior in integral-to-floating conversions

Section: 7.10 [conv.fpint] **Status:** CD2 **Submitter:** Alberto Ganesh Barbati **Date:** 2 Aug, 2008

[Voted into WP at July, 2009 meeting.]

The current wording of 7.10 [conv.fpint] paragraph 2 does not specify what should happen when converting an integer value that is outside the representable range of the target floating point type. The C99 Standard covers this case explicitly in 6.3.1.4 paragraph 2:

> When a value of integer type is converted to a real floating type, if the value being converted can be represented exactly in the new type, it is unchanged. If the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is either the nearest higher or nearest lower representable value, chosen in an implementation-defined manner. If the value being converted is outside the range of values that can be represented, the behavior is undefined.

While the current C++ specification requires defined behavior in all cases, the C specification allows for use of NaNs and traps, if those are needed for efficiency.

Notes from the September, 2008 meeting:

The CWG agreed that the C approach should be adopted.

Proposed resolution (March, 2009):

Change 7.10 [conv.fpint] paragraph 2 as indicated:

> An rvalue of an integer type or of an unscoped enumeration type can be converted to an rvalue of a floating point type. The result is exact if possible. ~~Otherwise~~ **If the value being converted is in the range of values that can be represented but cannot be represented exactly**, it is an implementation-defined choice of either the next lower or higher representable value. [*Note:* loss of precision occurs if the integral value cannot be represented exactly as a value of the floating type. —*end note*] **If the value being converted is outside the range of values that can be represented, the behavior is undefined.** If the source type is `bool`, the value `false` is converted to zero and the value `true` is converted to one.

---

438. Possible flaw in wording for multiple accesses to object between sequence points

Section: 8 [expr] **Status:** CD2 **Submitter:** Jason Merrill **Date:** 29 Oct 2003

Lisa Lippincott mentioned this case to me:

```
A[0] = 0;
A[A[0]] = 1;
```

This seems to use the old value of A[0] other than to calculate the new value, which is said to be undefined, but it also seems reasonable, since the old value is used in order to select the object to modify, so there's no ordering ambiguity.

Steve Adamczyk: the ordering rule referred to is in 8 [expr] paragraph 4.

Notes from the March 2004 meeting:

Clark Nelson mentions that the C committee may have done something on this.

Note (July, 2009):

This issue was resolved by the adoption of the "sequenced before" wording.

---

695. Compile-time calculation errors in constexpr functions

Section: 8 [expr] **Status:** CD2 **Submitter:** Mike Miller **Date:** 9 June, 2008

[Voted into WP at October, 2009 meeting.]

Evaluating an expression like `1/0` is intended to produce undefined behavior during the execution of a program but be ill-formed at compile time. The wording for this is in 8 [expr] paragraph 4:

> If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined, unless such an expression appears where an integral constant expression is required (8.20 [expr.const]), in which case the program is ill-formed.

The formulation "appears where an integral constant expression is required" is intended as an acceptable Standardese circumlocution for "evaluated at compile time," a concept that is not directly defined by the Standard. It is not clear that this formulation adequately covers constexpr functions.

Notes from the September, 2008 meeting:

The CWG felt that the concept of "compile-time evaluation" needs to be defined for use in discussing constexpr functions. There is a tension between wanting to diagnose errors at compile time versus not diagnosing errors that will not actually occur at runtime. In this context, a constexpr function might never be invoked, either in a constant expression context or at runtime, although the requirement that the expression in a constexpr function be a potential constant expression could be interpreted as triggering the provisions of 8 [expr] paragraph 4.

There are also contexts in which it is not known in advance whether an expression must be constant or not, notably in the initializer of a const integer variable, where the nature of the initializer determines whether the variable can be used in constant expressions or not. In such a case, it is not clear whether an erroneous expression should be considered ill-formed or simply non-constant (and thus subject to runtime undefined behavior, if it is ever evaluated). The consensus of the CWG was that an expression like `1/0` should simply be considered non-constant; any diagnostic would result from the use of the expression in a context requiring a constant expression.

Proposed resolution (July, 2009):

This issue is resolved by the resolution of issue 699.

---

835. Scoped enumerations and the "usual arithmetic conversions"

Section: 8 [expr] **Status:** CD2 **Submitter:** Beman Dawes **Date:** 5 March, 2009

[Voted into WP at October, 2009 meeting.]

A number of the operators described in clause 8 [expr] take operands of enumeration type, relying on the "usual arithmetic conversions" (8 [expr] paragraph 10) to convert them to an appropriate integral type. The assumption behind this pattern is invalid when one or more of the operands has a scoped enumeration type.

Each operator that accepts operands of enumeration type should be evaluated as to whether the operation makes sense for scoped enumerations (for example, it is probably a good idea to allow comparison of operands having the same scoped enumeration type and conditional expressions in which the second and third operands have the same scoped enumeration type) and, if so, create a special case. The usual arithmetic conversions should not be invoked for scoped enumeration types.

(See also issue 880.)

Proposed resolution (July, 2009):

1. Change 8 [expr] paragraph 10 as follows:

> ...This pattern is called the usual arithmetic conversions, which are defined as follows:

- **If either operand is of scoped enumeration type (10.2 [dcl.enum]), no conversions are performed, and if the other operand does not have the same type, the expression is ill-formed.**
- If either operand is of type `long double`...

2. Change 8.2.1 [expr.sub] paragraph 1 as follows:

   ...One of the expressions shall have the type "pointer to `T`" and the other shall have **unscoped** enumeration or integral type...

3. Change 8.3 [expr.unary] paragraphs 7-8 and 10 as follows:

   The operand of the unary `+` operator shall have arithmetic, **unscoped** enumeration, or pointer type...

   The operand of the unary `-` operator shall have arithmetic or **unscoped** enumeration type...

   The operand of `~` shall have integral or **unscoped** enumeration type...

4. Change 8.3.4 [expr.new] paragraph 6 as follows:

   ...The expression in a *noptr-new-declarator* shall be of integral type, **unscoped** enumeration type, or a class type for which a single non-explicit conversion function to integral or **unscoped** enumeration type exists (15.3 [class.conv]). If the expression...

5. Change 8.6 [expr.mul] paragraph 2 as follows:

   The operands of `*` and `/` shall have arithmetic or **unscoped** enumeration type; the operands of `%` shall have integral or **unscoped** enumeration type....

6. Change 8.7 [expr.add] paragraph 1-2 as follows:

   ...For addition, either both operands shall have arithmetic or **unscoped** enumeration type, or one operand shall be a pointer to a completely-defined effective object type and the other shall have integral or **unscoped** enumeration type.

   For subtraction, one of the following shall hold:

   - both operands have arithmetic or **unscoped** enumeration type; or
   - both operands are pointers to cv-qualified or cv-unqualified versions of the same completely-defined effective object type; or
   - the left operand is a pointer to a completely-defined effective object type and the right operand has integral or **unscoped** enumeration type.

7. Change 8.8 [expr.shift] paragraph 1 as follows:

   ...The operands shall be of integral or **unscoped** enumeration type...

8. Change 8.9 [expr.rel] paragraph 4 as follows:

   If both operands (after conversions) are of arithmetic **or enumeration** type, each of the operators shall yield `true` if the specified relationship is true and `false` if it is false.

9. Change 8.11 [expr.bit.and] paragraph 1 as follows:

   ...The operator applies only to integral or **unscoped** enumeration operands.

10. Change 8.12 [expr.xor] paragraph 1 as follows:

    ...The operator applies only to integral or **unscoped** enumeration operands.

11. Change 8.13 [expr.or] paragraph 1 as follows:

    ...The operator applies only to integral or **unscoped** enumeration operands.

---

858. Example binding an rvalue reference to an lvalue

Section: 8 [expr] **Status:** CD2 **Submitter:** Daniel Krügler **Date:** 6 April, 2009

[Voted into WP at March, 2010 meeting as part of document N3055.]

The adoption of paper N2844 made it ill-formed to attempt to bind an rvalue reference to an lvalue, but the example in 8 [expr] paragraph 6 was overlooked in making this change:

```
struct A { };
A&& operator+(A, A);
A&& f();

A a;
A&& ar = a;
```

The last line should be changed to use something like `static_cast<A&&>(a)`.

(See also issue 847.)

Proposed resolution (July, 2009):

Change the example in 8 [expr] paragraph 6 as follows:

> [*Example:*
>
> ```
> struct A { };
> A&& operator+(A, A);
> A&& f();
>
> A a;
> A&& ar = static_cast<A&&>(a);
> ```
>
> The expressions `f()` and `a + a` are rvalues of type `A`. The expression `ar` is an lvalue of type `A`. —*end example*]

---

720. Need examples of *lambda-expression*s

Section: 8.1.5 [expr.prim.lambda] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 20 September, 2008

N2800 comment DE 9

[Voted into the WP at the July, 2009 meeting as part of N2927.]

There is not a single example of a *lambda-expression* in their specification. The Standard would be clearer if a few judiciously-chosen examples were added.

Proposed resolution (July, 2009)

See document PL22.16/09-0117 = WG21 N2927.

---

750. Implementation constraints on reference-only closure objects

Section: 8.1.5 [expr.prim.lambda] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 10 December, 2008

N2800 comment US 73

[Voted into the WP at the July, 2009 meeting as part of N2927.]

Consider an example like:

```
void f(vector<double> vec) {
  double x, y, z;
  fancy_algorithm(vec, [&]() { /* use x, y, and z in various ways */ });
}
```

8.1.5 [expr.prim.lambda] paragraph 8 requires that the closure class for this lambda will have three reference members, and paragraph 12 requires that it be derived from `std::reference_closure`, implying two additional pointer members. Although 11.3.2 [dcl.ref] paragraph 4 allows a reference to be implemented without allocation of storage, current ABIs require that references be implemented as pointers. The practical effect of these requirements is that the closure object for this lambda expression will contain five pointers. If not for these requirements, however, it would be possible to implement the closure object as a single pointer to the stack frame, generating data accesses in the function-call operator as offsets relative to the frame pointer. The current specification is too tightly constrained.

Lawrence Crowl:

The original intent was that the reference members could be omitted from the closure object by an implementation. The problem we had was that we want the call to `f` in

```
extern f(std::reference_closure<void()>);
extern f(std::function<void()>);
f([&](){});
```

to unambiguously bind to the `reference_closure`; using `reference_closure` can be an order of magnitude faster than using `function`.

(See also issue 751.)

Proposed resolution (July, 2009)

See document PL22.16/09-0117 = WG21 N2927. (See also document PL22.16/09-0035 = WG21 N2845, which partially addressed this issue by the removal of `std::reference_clossure`.)

---

751. Deriving from closure classes

Section: 8.1.5 [expr.prim.lambda] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 11 December, 2008

[Voted into the WP at the July, 2009 meeting as part of N2927.]

During the discussion of issue 750, it was suggested that an implementation might be permitted to omit fields in the closure object of a lambda expression if the implementation does not need them to address the corresponding automatic variables. If permitted, this implementation choice might be visible to the program via inheritance. Consider:

```
void f() {
  int const N = 10;
  typedef decltype([&N](){}) F;
  struct X: F {
    void n() { float z[N]; } // Error?
  };
}
```

If it is implementation-defined or unspecified whether the reference member `F::N` will exist, then it is unknown whether the the reference to `N` in `X::n()` will be an error (because lookup finds `F::N`, which is private) or well-formed (because there is no `F::N`, so the reference is to the local automatic variable).

If implementations can omit fields, the implementation dependency might be addressed by either treating the lookup "as if" the fields existed, even if they are not present in the object layout, or by defining the names of the fields in the closure class to be unique identifiers, similar to the names of unnamed namespaces (10.3.1.1 [namespace.unnamed]).

Another suggestion was made that derivation from a closure class should be prohibited, at least for now. However, it was pointed out that inheritance is frequently used to give stateless function objects some state, suggesting a use case along the lines of:

```
template<class T> struct SomeState: T {
  // ...
};
template<class F, typename T< void algo(T functor, ...) {
  SomeState<T< state(functor);
  ...
}

... algo([](int a){ return 2*a; }) ...
```

Proposed resolution (July, 2009)

See document PL22.16/09-0117 = WG21 N2927.

---

752. Name lookup in nested *lambda-expression*s

Section: 8.1.5 [expr.prim.lambda] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 10 December, 2008

N2800 comment US 31

[Voted into the WP at the July, 2009 meeting as part of N2927.]

How does name binding work in nested *lambda-expression*s? For example,

```
void f1() {
  float v;
  [](){ return [v](){ return v; }; }
}

void f2() {
  float v;
  [v](){ return [v](){ return v; }; }
}
```

According to 8.1.5 [expr.prim.lambda] paragraph 3,

> A name in the *lambda-capture* shall be in scope in the context of the lambda expression, and shall be `this` or shall refer to a local variable or reference with automatic storage duration.

One possible interpretation is that the lambda expression in `f1` is ill-formed because `v` is used in the *compound-statement* of the outer lambda expression but does not appear in its effective capture set. However, the appearance of `v` in the inner *lambda-capture* is not a "use" in the sense of 6.2 [basic.def.odr] paragraph 2, because a *lambda-capture* is not an *expression*, and it's not clear whether the reference in the inner lambda expression's `return` expression should be considered a use of the automatic variable or of the member of the inner lambda expression's closure object.

Similarly, the lambda expression in `f2` could be deemed to be ill-formed because the reference to `v` in the inner lambda expression's *lambda-capture* would refer to the field of the outer lambda-expression's closure object, not to a local automatic variable; however, it's not clear whether the inner lambda expression should be evaluated *in situ* or as part of the generated `operator()` member of the outer lambda expression's closure object.

Proposed resolution (July, 2009)

See document PL22.16/09-0117 = WG21 N2927.

---

753. Array names in lambda capture sets

Section: 8.1.5 [expr.prim.lambda] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 10 December, 2008

[Voted into the WP at the July, 2009 meeting as part of N2927.]

The current specification does not adequately describe what happens when an array name is part of the effective capture set of a lambda expression. 8.1.5 [expr.prim.lambda] paragraph 13 says that the array member of the closure object is direct-initialized by the local array; however, 11.6 [dcl.init] paragraph 16 says that such an initialization is ill-formed. There are several possibilities for handling this problem:

1. This results in an array member of the closure object, which is initialized by copying each element, along the lines of 15.8 [class.copy] paragraph 8.
2. This results in a pointer member of the closure object, initialized to point to the first element of the array (i.e., the array lvalue decays to a pointer rvalue).
3. This is ill-formed.
4. This results in a reference-to-array member of the closure object, initialized to refer to the array, regardless of whether `&` was used or not.
5. This is ill-formed unless the capture is "by reference."

Proposed resolution (July, 2009)

See document PL22.16/09-0117 = WG21 N2927.

---

754. Lambda expressions in default arguments of block-scope function declarations

Section: 8.1.5 [expr.prim.lambda] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 10 December, 2008

[Voted into the WP at the July, 2009 meeting as part of N2927.]

Is a lambda expression permitted in a default argument expression for a block-scope function declaration? For example,

```
void g() {
  void f(std::reference_closure<void()> rc = []() {});
  f();
}
```

This was not discussed in either the Evolution Working Group nor in the Core Working Group, and it is possible that some of the same implementation difficulties that led to prohibiting use of automatic variables in such default argument expressions (11.3.6 [dcl.fct.default] paragraph 7) might also apply to closure objects, even though they are not automatic variables.

(See also issue 772.)

Proposed resolution (July, 2009)

See document PL22.16/09-0117 = WG21 N2927.

---

756. Dropping cv-qualification on members of closure objects

Section: 8.1.5 [expr.prim.lambda] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 15 December, 2008

[Voted into the WP at the July, 2009 meeting as part of N2927.]

Consider the following example:

```
void f() {
  int const N = 10;
  [=]() mutable { N = 30; }  // Okay: this->N has type int, not int const.
  N = 20;  // Error.
}
```

That is, the `N` that is a member of the closure object is not const, even though the captured variable is const. This seems strange, as capturing is basically a means of capturing the local environment in a way that avoids lifetime issues. More seriously, the change of type means that the results of `decltype`, overload resolution, and template argument deduction applied to a captured variable inside a lambda expression can be different from those in the scope containing the lambda expression, which could be a subtle source of bugs.

On the other hand, the copying involved in capturing has uses beyond avoiding lifetime issues (taking snapshots of values, avoiding data races, etc.), and the value of a cv-qualified object is not cv-qualified.

Proposed resolution (July, 2009)

See document PL22.16/09-0117 = WG21 N2927.

---

759. Destruction of closure objects

Section: 8.1.5 [expr.prim.lambda] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 22 January, 2009

N2800 comment UK 39

[Voted into the WP at the July, 2009 meeting as part of N2927.]

The specification of closure objects is missing a couple of important points regarding their destruction. First, although 8.1.5 [expr.prim.lambda] paragraph 11 mentions other implicitly-declared special member functions, it is silent on the destructor, leading to questions about whether the closure class has one or not.

Second, nothing is said about the timing of the destruction of a closure object: is it normally destroyed at the end of the full-expression to which the lambda expression belongs, and is its lifetime extended if the closure object is bound to a reference? These questions would be addressed if paragraph 2 defined the closure object as a temporary instead of just as an rvalue. (It should be noted that 8.2.3 [expr.type.conv] also does not define the conceptually-similar `T()` as a temporary.)

Proposed resolution (July, 2009)

See document PL22.16/09-0117 = WG21 N2927. (The question regarding the failure of 8.2.3 [expr.type.conv] failing to categorize `T()` as a temporary was split off into a separate issue; see issue 943.)

---

761. Inferred return type of closure object call operator

Section: 8.1.5 [expr.prim.lambda] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 5 February, 2009

[Voted into the WP at the July, 2009 meeting as part of N2927.]

According to 8.1.5 [expr.prim.lambda] paragraph 10, the following lambda expressions are ill-formed because the return types of the generated `operator()` functions are an array type and a function type, respectively:

```
void f() {
  []{ return ""; };
  []{ return f; };
}
```

It would seem reasonable to expect the array-to-pointer and function-to-pointer decay to apply to these return values and hence to the inferred return type of `operator()`.

Proposed resolution (July, 2009)

See document PL22.16/09-0117 = WG21 N2927.

---

762. Name lookup in the *compound-statement* of a lambda expression

Section: 8.1.5 [expr.prim.lambda] **Status:** CD2 **Submitter:** John Spicer **Date:** 5 February, 2009

N2800 comment US 29
N2800 comment US 30

[Voted into the WP at the July, 2009 meeting as part of N2927.]

The current wording of 8.1.5 [expr.prim.lambda] is not clear as to how name lookup is to be performed for names appearing in the *compound-statement* of a lambda expression. Consider, for example:

```
int fac(int n) {
  return [=]{ return n <= 1 ? 1 : n*operator()(n-1); }();
}
```

There is no `operator()` in scope in the context of the lambda expression. Consequently, according to bullet 5 of paragraph 10, the reference to `operator()` is not transformed to the class member access syntax but appears untransformed in the closure object's function call operator, where presumably it is interpreted as a recursive call to itself.

A similar question (although it does not involve name lookup per se) arises with respect to use of `this` in the *compound-statement* of a lambda expression that does not appear in the body of a non-static member function; for example,

```
void f() {
  float v;
  [v]() { return v+this->v; }
}
```

`this` cannot refer to anything except the closure object, so are the two references to `v` equivalent?

The crux of this question is whether the lookups for names in the *compound-statement* are done in the context of the lambda expression or from the call operator of the closure object. The note at the end of paragraph 10 bullet 5 would tend to support the latter interpretation:

> [*Note:* Reference to captured variables or references within the *compound-statement* refer to the data members of `F`. —*end note*]

Another possible interpretation of the current wording is that there are two distinct *compound-statement*s in view: the *compound-statement* that is part of the *lambda-expression* and the body of the closure object's function call operator that is "obtained from" the former. If this is the intended interpretation, one way of addressing the issues regarding the `operator()` example above would be to state that it is an error if the lookup of a name in the *compound-statement* fails, making the example ill-formed.

Proposed resolution (July, 2009)

See document PL22.16/09-0117 = WG21 N2927.

---

763. Is a closure object's `operator()` inline?

Section: 8.1.5 [expr.prim.lambda] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 6 February, 2009

[Voted into the WP at the July, 2009 meeting as part of N2927.]

A lambda expression appearing in local scope presumably creates a local class (in the sense of 12.4 [class.local]) as the type of the closure object, because that class is "considered to be defined at the point where the lambda expression occurs" (8.1.5 [expr.prim.lambda] paragraph 7), and in the absence of any indication to the contrary that class must satisfy the restrictions of 12.4 [class.local] on local classes. One such restriction is that all its member functions must be defined within the class definition, making them inline. However, nothing is said about whether the function call operator for a non-local closure class is inline, and even for the local case it would be better if the specification were explicit.

Proposed resolution (July, 2009)

See document PL22.16/09-0117 = WG21 N2927.

---

764. Capturing unused variables in a lambda expression

Section: 8.1.5 [expr.prim.lambda] **Status:** CD2 **Submitter:** Steve Adamczyk **Date:** 6 February, 2009

[Voted into the WP at the July, 2009 meeting as part of N2927.]

8.1.5 [expr.prim.lambda] paragraph 5 says,

> The *compound-statement* of a lambda expression shall use (6.2 [basic.def.odr]) an automatic variable or reference from the context where the lambda expression appears only if the name of the variable or reference is a member of the effective capture set...

The reference to 6.2 [basic.def.odr] makes clear that the technical meaning of "use" is in view here, and that the names of variables can appear without being captured if they are constants used as values or if they are unevaluated operands.

There appears to be a disconnect with the preceding paragraph, however, in the description of which variables are implicitly captured by a *capture-default*.

> for each name `v` that appears in the lambda expression and denotes a local variable or reference with automatic storage duration in the context where the lambda expression appears and that does not appear in the *capture-list* or as a parameter name in the *lambda-parameter-declaration-list*...

It would be more consistent if only variables that were required by paragraph 5 to be captured were implicitly captured, i.e., if "that appears in the lambda expression" were replaced by "that is used (6.2 [basic.def.odr]) in the *compound-statement* of the lambda expression." For example,

```
struct A {
  A();
  A(const A&);
  ~A();
};
void f() {
  A a;
  int i = [=]() { return sizeof a; }();
}
```

Here, `a` will be captured (and copied), even though it is not "used" in the lambda expression.

Proposed resolution (July, 2009)

See document PL22.16/09-0117 = WG21 N2927.

---

766. Where may lambda expressions appear?

Section: 8.1.5 [expr.prim.lambda] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 6 February, 2009

[Voted into the WP at the July, 2009 meeting as part of N2927.]

According to 8.1.5 [expr.prim.lambda] paragraph 7, the appearance of a lambda expression results in the definition of a class "considered to be defined at the point where the lambda expression occurs." It is not clear whether that means that a lambda expression cannot appear at any point where it is not permitted to define a class type. For example, 11.3.5 [dcl.fct] paragraph 10 says, "Types shall not be defined in return or parameter types." Does that mean that a function declaration like

```
void f(int a[sizeof ([]{ return 0; })]);
```

is ill-formed, because the parameter type defines the closure class for the lambda expression? (Issue 686 lists many contexts in which type definitions are prohibited. Each of these should be examined to see whether a lambda expression should be allowed or prohibited there.)

Proposed resolution (July, 2009)

See document PL22.16/09-0117 = WG21 N2927.

---

767. `void` **and other unnamed *lambda-parameter*s**

Section: 8.1.5 [expr.prim.lambda] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 5 February, 2009

[Voted into the WP at the July, 2009 meeting as part of N2927.]

The grammar in 8.1.5 [expr.prim.lambda] for *lambda-parameter* specifies that a *declarator* must be present, i.e., that all *lambda-parameter*s must be named. This also has the effect of prohibiting a lambda like `[](void){}`. It is not clear that there is a good reason for these restrictions; programmers could reasonably expect that *lambda-parameter*s were like ordinary function parameters in these regards.

Proposed resolution (July, 2009)

See document PL22.16/09-0117 = WG21 N2927.

---

768. Ellipsis in a lambda parameter list

Section: 8.1.5 [expr.prim.lambda] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 5 February, 2009

[Voted into the WP at the July, 2009 meeting as part of N2927.]

The grammar in 8.1.5 [expr.prim.lambda] for *lambda-parameter-declaration* does not allow for an ellipsis. Is this a desirable restriction?

Proposed resolution (July, 2009)

See document PL22.16/09-0117 = WG21 N2927.

---

769. Initialization of closure objects

Section: 8.1.5 [expr.prim.lambda] **Status:** CD2 **Submitter:** Mike Miller **Date:** 9 February, 2009

[Voted into the WP at the July, 2009 meeting as part of N2927.]

8.1.5 [expr.prim.lambda] paragraph 13 says simply,

> The closure object is initialized by direct-initializing each member `N` of `F` with the local variable or reference named `N`; the member `t` is initialized with `this`.

The mechanism for this initialization is not specified. In particular, does the closure class have a default constructor that performs this initialization?

Proposed resolution (July, 2009)

See document PL22.16/09-0117 = WG21 N2927.

---

771. Move-construction of reference members of closure objects

Section: 8.1.5 [expr.prim.lambda] **Status:** CD2 **Submitter:** Jonathan Caves **Date:** 9 February, 2009

[Voted into the WP at the July, 2009 meeting as part of N2927.]

According to 8.1.5 [expr.prim.lambda] paragraph 11, the closure class "has a public move constructor that performs a member-wise move." Although the terms "move constructor" and "member-wise move" are not currently defined (see issue 680), this presumably means that a lambda like `[&i]{}` results in a closure class similar to:

```
class F {
    int& i;
public:
    F(&& other):
        i(std::move(other.i)) { }
    // etc.
};
```

This constructor is ill-formed because it attempts to initialize an lvalue reference to non-const int with the rvalue returned by `std::move`.

It is not clear whether this should be handled by:

1. Not generating the move constructor.
2. Generating the declaration of the move constructor but only defining it (and giving the corresponding error) if the move constructor would be used, similar to the handling of other implicitly-defined special member functions.
3. Generating the move constructor but copy-constructing any reference members.

Proposed resolution (July, 2009)

See document PL22.16/09-0117 = WG21 N2927.

---

772. *capture-default* in lambdas in local default arguments

Section: 8.1.5 [expr.prim.lambda] **Status:** CD2 **Submitter:** Steve Adamczyk **Date:** 10 February, 2009

[Voted into the WP at the July, 2009 meeting as part of N2927.]

Assuming that it is permitted to use a lambda as a default argument in a block-scope function declaration (see issue 754), it is presumably ill-formed for such a lambda expression to refer to a local automatic variable (11.3.6 [dcl.fct.default] paragraph 7). What does this mean for *capture-default*s? For example,

```
void f() {
  int i = 1;
  void f(int = ([i]() { return i; })());  // Definitely an error
  void g(int = ([i]() { return 0; })());  // Probably an error
  void h(int = ([=]() { return i; })());  // Definitely an error
  void o(int = ([=]() { return 0; })());  // Okay?
  void p(int = ([]() { return sizeof i; })());  // Presumably okay
}
```

Proposed resolution (July, 2009)

See document PL22.16/09-0117 = WG21 N2927.

---

774. Can a closure class be a POD?

Section: 8.1.5 [expr.prim.lambda] **Status:** CD2 **Submitter:** John Spicer **Date:** 11 February, 2009

[Voted into the WP at the July, 2009 meeting as part of N2927.]

The current wording does not state under what conditions, if ever, a closure class is a POD. It should either be explicitly unspecified or definitively stated that a closure class is never a POD, to allow implementations freedom to determine the contents of closure classes.

Notes from the March, 2009 meeting:

A closure class is neither standard-layout nor trivial.

Proposed resolution (July, 2009)

See document PL22.16/09-0117 = WG21 N2927.

---

775. Capturing references to functions

Section: 8.1.5 [expr.prim.lambda] **Status:** CD2 **Submitter:** Steve Adamczyk **Date:** 12 February, 2009

[Voted into the WP at the July, 2009 meeting as part of N2927.]

According to 8.1.5 [expr.prim.lambda] paragraph 8, the "object type" of a captured function is the type to which the reference refers. That's clearly wrong when the captured reference is a reference to a function, because the resulting data member of the closure class will have a function type:

```
void f() { }
void g() {
  void (&fr)() = f;
  [fr]{};   // Oops...
}
```

Proposed resolution (July, 2009)

See document PL22.16/09-0117 = WG21 N2927.

---

779. Rvalue reference members of closure objects?

Section: 8.1.5 [expr.prim.lambda] **Status:** CD2 **Submitter:** Mike Miller **Date:** 26 February, 2009

[Voted into the WP at the July, 2009 meeting as part of N2927.]

8.1.5 [expr.prim.lambda] paragraph 8, bullet 2, says of members of a closure class,

> if the element is of the form `& N`, the data member has the name `N` and type "reference to object type of `N`"

Is an implementation free to use an rvalue reference as the type of this member, as only a "reference" is specified? (See issue 771; the move constructor would be well-formed if the reference member were an rvalue reference.)

Proposed resolution (July, 2009)

See document PL22.16/09-0117 = WG21 N2927.

---

782. Lambda expressions and argument-dependent lookup

Section: 8.1.5 [expr.prim.lambda] **Status:** CD2 **Submitter:** Mike Miller **Date:** 1 March, 2009

[Voted into the WP at the July, 2009 meeting as part of N2927.]

Functions and function objects behave differently with respect to argument-dependent lookup. In particular, the associated namespaces of a function's parameters and return types, but not the namespace in which the function is declared, are associated namespaces of the function; the exact opposite is true of a function object. The Committee should consider rectifying that disparity; however, in the absence of such action, an explicit decision should be made as to whether lambdas are more function-like or object-like with respect to argument-dependent lookup. For example:

```
namespace M {
  struct S { };
}
namespace N {
  void func(M::S);
  struct {
    void operator()(M::S);
  } fn_obj;
  const auto& lambda = [](M::S){};
}
void g() {
  f(N::func);    // assoc NS == M, not N
  f(N::fn_obj);  // assoc NS == N, not M
  f(N::lambda);  // assoc NS == ??
}
```

Proposed resolution (July, 2009)

See document PL22.16/09-0117 = WG21 N2927.

---

796. Lifetime of a closure object with members captured by reference

Section: 8.1.5 [expr.prim.lambda] **Status:** CD2 **Submitter:** UK **Date:** 3 March, 2009

N2800 comment UK 40

8.1.5 [expr.prim.lambda] paragraph 13 ties the effective lifetime of a closure object with members captured by reference to the innermost block scope in which the lambda appears, rather than to the lifetime of the objects to which the references are bound. This seems too restrictive.

Notes from the March, 2009 meeting:

Making the suggested change would be problematic for an implementation in which the "reference members" were actually implemented using offsets from a captured stack pointer and in which nested blocks were pushed onto the stack (to optimize space for large local objects, for example).

Proposed resolution (July, 2009)

See document PL22.16/09-0117 = WG21 N2927.

---

797. Converting a no-capture lambda to a function type

Section: 8.1.5 [expr.prim.lambda] **Status:** CD2 **Submitter:** UK **Date:** 3 March, 2009

N2800 comment UK 42

[Voted into WP at March, 2010 meeting as document N3052.]

A lambda with an empty capture list has identical semantics to a regular function type. By requiring this mapping we get an efficient lambda type with a known API that is also compatible with existing operating system and C library functions.

Notes from the July, 2009 meeting:

This functionality is part of the "unified function syntax" proposal and will be considered in that context.

---

904. Parameter packs in *lambda-capture*s

Section: 8.1.5 [expr.prim.lambda] **Status:** CD2 **Submitter:** Faisal Vali **Date:** 23 May, 2009

[Voted into WP at March, 2010 meeting.]

The following is not allowed by the current syntax of *lambda-capture* but would be useful:

```
template <typename ...Args> void f(Args... args) {
  auto l = [&, args...] { return g(args...); };
}
```

Proposed resolution (October, 2009):

1. Change the grammar in 8.1.5 [expr.prim.lambda] paragraph 1 as follows:

   capture-list:
   capture . . . ***opt***
   capture-list , *capture* . . . ***opt***

2. Add a new paragraph at the end of 8.1.5 [expr.prim.lambda]:

   A *capture* followed by an ellipsis is a pack expansion (17.6.3 [temp.variadic]). [*Example:*

   ```
   template<typename ...Args>
   void f(Args... args) {
     auto l = [&, args...] { return g(args...); };
     l();
   }
   ```

   ***—end example*]**

3. Add a new bullet to the list in 17.6.3 [temp.variadic] paragraph 4:

   - **In a *capture-list* (8.1.5 [expr.prim.lambda]); the pattern is a *capture*.**

[Editorial note: the editor may wish to consider sorting the bullets in this list in order of section reference.]

---

955. Can a closure type's `operator()` be virtual?

Section: 8.1.5 [expr.prim.lambda] **Status:** CD2 **Submitter:** Daniel Krügler **Date:** 19 August, 2009

[Voted into WP at March, 2010 meeting.]

The specification of the members of a closure type does not rule out the possibility that its `operator()` might be virtual. It would be better to make it clear that it cannot.

Proposed resolution (October, 2009):

Change 8.1.5 [expr.prim.lambda] paragraph 5 as follows:

> ... followed by `mutable`. It is ~~not~~ **neither virtual nor** declared `volatile`. Default arguments...

---

863. Rvalue reference cast to incomplete type

Section: 8.2 [expr.post] **Status:** CD2 **Submitter:** Steve Adamczyk **Date:** 7 April, 2009

[Voted into WP at March, 2010 meeting as document N3055.]

A cast to an rvalue reference type produces an rvalue, and rvalues must have complete types (6.10 [basic.lval] paragraph 9). However, none of the sections dealing with cast operators in 8.2 [expr.post] require that the referred-to type must be complete in an rvalue reference cast.

(Note that the approach described for issue 690, in which an rvalue reference type would be essentially an lvalue instead of an rvalue, would address this issue as well, since lvalues can have incomplete types.)

Proposed resolution (February, 2010):

See paper N3030.

---

722. Can `nullptr` be passed to an ellipsis?

Section: 8.2.2 [expr.call] **Status:** CD2 **Submitter:** Alisdair Meredith **Date:** 25 September, 2008

[Voted into WP at March, 2010 meeting.]

The current wording of 8.2.2 [expr.call] paragraph 7 is:

> After these conversions, if the argument does not have arithmetic, enumeration, pointer, pointer to member, or effective class type, the program is ill-formed.

It's not clear whether this is intended to exclude anything other than `void`, but the effect is to disallow passing `nullptr` to ellipsis. That seems unnecessary.

Notes from the October, 2009 meeting:

The CWG agreed that this should be supported and the effect should be like passing `(void*)nullptr`.

Proposed resolution (February, 2010):

Change 8.2.2 [expr.call] paragraph 7 as follows:

> When there is no parameter for a given argument, the argument is passed in such a way that the receiving function can obtain the value of the argument by invoking `va_arg` (21.10 [support.runtime]). [*Note:* This paragraph does not apply to arguments passed to a function parameter pack. Function parameter packs are expanded during template instantiation (17.6.3 [temp.variadic]), thus each such argument has a corresponding parameter when a function template specialization is actually called. —*end note*] The lvalue-to-rvalue (7.1 [conv.lval]), array-to-pointer (7.2 [conv.array]), and function-to-pointer (7.3 [conv.func]) standard conversions are performed on the argument expression. **An argument that has (possibly cv-qualified) type `std::nullptr_t` is converted to type `void*` (7.11 [conv.ptr]).** After these conversions...

---

731. Omitted reference qualification of member function type

Section: 8.2.5 [expr.ref] **Status:** CD2 **Submitter:** Daniel Krügler **Date:** 6 October, 2008

N2800 comment DE 5
N2800 comment DE 10
N2800 comment DE 12

[Voted into WP at October, 2009 meeting.]

There are several places in the Standard that were overlooked when reference qualification of member functions was added. For example, 8.2.5 [expr.ref] paragraph 4, bullet 3, sub-bullet 2 says,

> ...if `E1.E2` refers to a non-static member function, and the type of `E2` is "function of parameter-type-list *cv* returning `T`", then...

This wording incorrectly excludes member functions declared with a *ref-qualifier*.

Another place that should consider reference qualification is 8.5 [expr.mptr.oper]; it should not be possible to invoke an &-qualified member function with an rvalue object expression.

A third place is 10.3.3 [namespace.udecl] paragraph 15, which does not mention reference qualification in connection with the hiding/overriding of member functions brought in from a base class via a *using-declaration*.

Proposed resolution (September, 2009):

1. Change 8.2.5 [expr.ref] paragraph 4, bullet 3, sub-bullet 2 as follows:
   - ...
     - ...
     - Otherwise, if `E1.E2` refers to a non-static member function and the type of `E2` is "function of parameter-type-list *cv* ***ref-qualifier*$_{opt}$** returning `T`", then `E1.E2` is an rvalue. The expression designates a non-static member function...
2. Change 8.5 [expr.mptr.oper] paragraph 6 as follows:

   ...—*end example*] **In a `.*` expression whose object expression is an rvalue, if the second operand is a pointer to member function with *ref-qualifier* `&`, the program is ill-formed. In a `->*` expression, or in a `.*` expression whose object expression is an lvalue, if the second operand is a pointer to member function with *ref-qualifier* `&&`, the program is ill-formed.** The result of a `.*` expression is an lvalue only if its first operand is an lvalue and...
3. Change 10.3.3 [namespace.udecl] paragraph 15 as follows:

   When a *using-declaration* brings names from a base class into a derived class scope, member functions and member function templates in the derived class override and/or hide member functions and member function templates with the same name, parameter-type-list (11.3.5 [dcl.fct]), ~~and~~ cv-qualification**, and *ref-qualifier* (if any)** in a base class (rather than conflicting). [*Note*:...

---

665. Problems in the specification of `dynamic_cast`

Section: 8.2.7 [expr.dynamic.cast] **Status:** CD2 **Submitter:** Daniel Krügler **Date:** 1 December 2007

[Voted into the WP at the March, 2009 meeting.]

At least one implementation accepts the following example as well-formed (returning a null pointer at runtime), although others reject it at compile time:

```
struct A { virtual ~A(); };
struct B: private A { } b;
A* pa = dynamic_cast<A*>(&b);
```

Presumably the intent of 8.2.7 [expr.dynamic.cast] paragraph 5 is that all up-casts (converting from derived to base) are to be handled at compile time, regardless of whether the class involved is polymorphic or not:

> If `T` is "pointer to *cv1* `B`" and `v` has type "pointer to *cv2* `D`" such that `B` is a base class of `D`, the result is a pointer to the unique `B` subobject of the `D` object pointed to by `v`. Similarly, if `T` is "reference to *cv1* `B`" and `v` has type *cv2* `D` such that `B` is a base class of `D`, the result is the unique `B` subobject of the `D` object referred to by `v`... In both the pointer and reference cases, *cv1* shall be the same cv-qualification as, or greater cv-qualification than, *cv2*, and `B` shall be an accessible unambiguous base class of `D`.

One explanation for the implementation that accepts the example at compile time is that the final sentence is interpreted as part of the condition for the applicability of this paragraph, so that this case falls through into the description of runtime checking that follows. This (mis-)interpretation is buttressed by the example in paragraph 9, which reads in significant part:

```
class A { virtual void f(); };
class B { virtual void g(); };
class D : public virtual A, private B {};
void g() {
    D d;
    B* bp;
    bp = dynamic_cast<B*>(&d); // fails
}
```

The "fails" comment is identical to the commentary on the lines in the example where the run-time check fails. If the interpretation that paragraph 5 is supposed to apply to all up-casts, presumably this comment should change to "ill-formed," or the line should be removed from the example altogether.

It should be noted that this interpretation (that the example is ill-formed and the runtime check applies only to down-casts and cross-casts) rejects some programs that could plausibly be accepted and actually work at runtime. For example,

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

void test(D* pd) {
    B* pb = dynamic_cast<B*>(pd); // #1
}

struct D2: virtual B, virtual D {};

void demo() {
    D2 d2;
    B* pb = dynamic_cast<B*>(&d2); // #2
    test(&d2); // #3
}
```

According to the interpretation that paragraph 5 applies, line #1 is ill-formed. However, converting from `D2` to `B` (line #2) is well-formed; if the alternate interpretation were applied, the conversion in line #1 could succeed when applied to `d2` (line #3).

One final note: the wording in 8.2.7 [expr.dynamic.cast] paragraph 8 is incorrect:

> The run-time check logically executes as follows:
>
> - If, in the most derived object pointed (referred) to by `v`, `v` points (refers) to a `public` base class subobject of a `T` object, and if only one object of type `T` is derived from the subobject pointed (referred) to by `v` the result is a pointer (an lvalue referring) to that `T` object.
> - Otherwise, if `v` points (refers) to a `public` base class subobject of the most derived object, and the type of the most derived object has a base class, of type `T`, that is unambiguous and `public`, the result is a pointer (an lvalue referring) to the `T` subobject of the most derived object.
> - Otherwise, the run-time check fails.

All uses of `T` in this paragraph treat it as if it were a class type; in fact, `T` is the type to which the expression is being cast and thus is either a pointer type or a reference type, not a class type.

Proposed resolution (June, 2008):

1. Change 8.2.7 [expr.dynamic.cast] paragraph 5 as follows:

   > ...In both the pointer and reference cases, ~~*cv1* shall be the same cv-qualification as, or greater cv-qualification than, *cv2*, and `B` shall be an accessible unambiguous base class of `D`~~ **the program is ill-formed if *cv2* is greater cv-qualification than *cv1* or if `B` is an inaccessible or ambiguous base class of `D`**.

2. Change the comment in the example in 8.2.7 [expr.dynamic.cast] paragraph 9 as follows:

   ```
   bp = dynamic_cast<B*>(&d);      // ~~fails~~ ill-formed (not a run-time check)
   ```

3. Change 8.2.7 [expr.dynamic.cast] paragraph 8 as follows:

   > ~~The~~ **If `C` is the class type to which `T` points or refers, the** run-time check logically executes as follows:
   >
   > - If, in the most derived object pointed (referred) to by `v`, `v` points (refers) to a public base class subobject of a `T` `C` object, and if only one object of type `T` `C` is derived from the subobject pointed (referred) to by `v` the result is a pointer (an lvalue referring) to that `T` `C` object.
   > - Otherwise, if `v` points (refers) to a public base class subobject of the most derived object, and the type of the most derived object has a base class, of type `T` `C`, that is unambiguous and public, the result is a pointer (an lvalue referring) to the `T` `C` subobject of the most derived object.
   > - Otherwise, the run-time check fails.

---

833. Explicit conversion of a scoped enumeration value to a floating type

Section: 8.2.9 [expr.static.cast] **Status:** CD2 **Submitter:** John Spicer **Date:** 6 March, 2009

[Voted into WP at October, 2009 meeting.]

The current wording of 8.2.9 [expr.static.cast] paragraph 9 does not permit conversion of a value of a scoped enumeration type to a floating point type. This was presumably an oversight during the specification of scoped enumerations and should be rectified.

Proposed resolution (July, 2009):

Change 8.2.9 [expr.static.cast] paragraph 9 as follows:

> A value of a scoped enumeration type (10.2 [dcl.enum]) can be explicitly converted to an integral type. The value is unchanged if the original value can be represented by the specified type. Otherwise, the resulting value is unspecified. **A value of a scoped enumeration type can also be explicitly converted to a floating point type; the result is the same as that of converting from the original value to the floating point type.**

---

658. Defining `reinterpret_cast` for pointer types

Section: 8.2.10 [expr.reinterpret.cast] **Status:** CD2 **Submitter:** Dave Abrahams **Date:** 4 November 2007

[Voted into the WP at the March, 2009 meeting.]

For years I've noticed that people will write code like this to get the address of an object's bytes:

```
void foo(long* p) {
    char* q = reinterpret_cast<char*>(p);  // #1
    // do something with the bytes of *p by using q
}
```

When in fact the only portable way to do it according to the standard is:

```
void foo(long* p) {
    char* q = static_cast<char*>(static_cast<void*>(p));  // #2
    // do something with the bytes of *p by using q
}
```

I thought `reinterpret_cast` existed so that vendors could provide some weird platform-specific things. However, recently Peter Dimov pointed out to me that if we substitute a class type for `long` above, `reinterpret_cast` is required to work as expected by 12.2 [class.mem] paragraph 18:

> A pointer to a standard-layout struct object, suitably converted using a `reinterpret_cast`, points to its initial member (or if that member is a bit-field, then to the unit in which it resides) and vice versa.

So there isn't a whole lot of flexibility to do something different and useful on non-class types. Are there any implementations for which #1 actually fails? If not, I think it would be a good idea to nail `reinterpret_cast` down so that the standard says it does what people (correctly) think it does in practice.

Proposed resolution (March, 2008):

Change 8.2.10 [expr.reinterpret.cast] paragraph 7 as indicated:

> A pointer to an object can be explicitly converted to a pointer to an object of different type. **When an rvalue `v` of type "pointer to `T1`" is converted to the type "pointer to *cv* `T2`," the result is `static_cast<`*cv* `T2*>(static_cast<`*cv* `void*>(v))` if both `T1` and `T2` are standard-layout types (6.9 [basic.types]) and the alignment requirements of `T2` are no stricter than those of `T1`.** ~~Except that c~~**C**onverting an rvalue of type "pointer to `T1`" to the type "pointer to `T2`" (where `T1` and `T2` are object types and where the alignment requirements of `T2` are no stricter than those of `T1`) and back to its original type yields the original pointer value~~, t~~**. T**he result of **any other** such ~~a~~ pointer conversion is unspecified.

---

734. Are unique addresses required for namespace-scope variables?

Section: 8.2.10 [expr.reinterpret.cast] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 15 October, 2008

[Voted into WP at March, 2010 meeting.]

Consider the following example:

```
static const char test1 = 'x';
static const char test2 = 'x';
bool f() {
    return &test1 != &test2;
}
```

Is `f()` allowed to return `false`? Can a smart optimizer alias these two variables, taking advantage of the fact that they are `const`, initialized to the same value, and thus can never be different in a well-defined program?

The C++ Standard doesn't explicitly specify address allocation of objects except as members of arrays and classes, so the answer would appear to be that such an implementation would be conforming.

This situation appears to have been the inadvertent result of the resolution of issue 73. Prior to that change, 8.10 [expr.eq] said,

> Two pointers of the same type compare equal if and only if they... both point to the same object...

That resolution introduced the current wording,

> Two pointers of the same type compare equal if and only if... both represent the same address.

Notes from the March, 2009 meeting:

The CWG agreed that this aliasing should not be permitted in a conforming implementation.

Proposed resolution (November, 2009):

1. Add the following as a new paragraph after 4.5 [intro.object] paragraph 5:

**Unless an object is a bit-field or a base class subobject of zero size, the address of that object is the address of the first byte it occupies. Two distinct objects that are neither bit-fields nor base class subobjects of zero size shall have distinct addresses [*Footnote:* Under the "as-if" rule an implementation is allowed to store two objects at the same machine address or not store an object at all if the program cannot observe the difference (4.6 [intro.execution]). —*end footnote*]. [*Example:*

```
static const char test1 = 'x';
static const char test2 = 'x';
const bool b = &test1 != &test2;    // always true
```

—*end example*]**

2. Change 8.3.1 [expr.unary.op] paragraph 3 as follows:

The result of the unary `&` operator is a pointer to its operand. The operand shall be an lvalue or a *qualified-id*. ~~In the first case, if the type of the expression is "T," the type of the result is "pointer to T." In particular, the address of an object of type "cv T" is "pointer to cv T," with the same cv-qualifiers. For a *qualified-id*, if the member is a static member of type "T", the type of the result is plain "pointer to T." If the member is a non-static member of class C of type T, the type of the result is "pointer to member of class C of type T."~~ **If the operand is a *qualified-id* naming a non-static member `m` of some class `C` with type `T`, the result has type "pointer to member of class `C` of type `T`" and is an rvalue designating `C::m`. Otherwise, if the type of the expression is `T`, the result has type "pointer to `T`" and is an rvalue that is the address of the designated object (4.4 [intro.memory]) or a pointer to the designated function. [*Note:* In particular, the address of an object of type "*cv* `T`" is "pointer to *cv* `T`," with the same cv-qualification. —*end note*]** [*Example:*...

---

799. Can `reinterpret_cast` be used to cast an operand to its own type?

Section: 8.2.10 [expr.reinterpret.cast] **Status:** CD2 **Submitter:** UK **Date:** 3 March, 2009

N2800 comment UK 55

[Voted into WP at March, 2010 meeting.]

The note in 8.2.10 [expr.reinterpret.cast] paragraph 2 says,

> Subject to the restrictions in this section, an expression may be cast to its own type using a `reinterpret_cast` operator.

However, there is nothing in the normative text that permits this conversion, and paragraph 1 forbids any conversion not explicitly permitted.

(See also issue 944.)

Proposed resolution (October, 2009):

1. Change 8.2.10 [expr.reinterpret.cast] paragraph 2 as follows:

The `reinterpret_cast` operator shall not cast away constness (8.2.11 [expr.const.cast]). ~~[*Note:* Subject to the restrictions in this section, an expression may be cast to its own type using a `reinterpret_cast` operator. —*end note*]~~ **An expression of integral, enumeration, pointer, or pointer-to-member type can be explictly converted to its own type; such a cast yields the value of its operand.**

2. Change 8.2.10 [expr.reinterpret.cast] paragraph 10 as follows:

An rvalue of type "pointer to member of `X` of type `T1`" can be explicitly converted to an rvalue of **a different** type "pointer to member of `Y` of type `T2`" if `T1` and `T2` are both function types or both object types...

---

842. Casting to rvalue reference type

Section: 8.2.10 [expr.reinterpret.cast] **Status:** CD2 **Submitter:** Steve Adamczyk **Date:** 20 March, 2009

[Voted into WP at October, 2009 meeting.]

Both `const_cast` (8.2.11 [expr.const.cast] paragraph 1) and `reinterpret_cast` (8.2.10 [expr.reinterpret.cast] paragraph 1) say,

> If `T` is an lvalue reference type, the result is an lvalue; otherwise, the result is an rvalue and the lvalue-to-rvalue (7.1 [conv.lval]), array-to-pointer (7.2 [conv.array]), and function-to-pointer (7.3 [conv.func]) standard conversions are performed on the expression `v`.

This introduces a contradiction in the text. According to 8.2.11 [expr.const.cast] paragraph 4,

> The result of a reference `const_cast` refers to the original object.

However, the lvalue-to-rvalue conversion applied to the operand when the target is an rvalue reference type creates a temporary if the operand has class type (7.1 [conv.lval] paragraph 2), meaning that the result will **not** refer to the original object but to the temporary.

A similar problem exists for `reinterpret_cast`: according to 8.2.10 [expr.reinterpret.cast] paragraph 11,

> a reference cast `reinterpret_cast<T&>(x)` has the same effect as the conversion `*reinterpret_cast<T*>(&x)` with the built-in `&` and `*` operators (and similarly for `reinterpret_cast<T&&>(x)`). The result refers to the same object as the source lvalue, but with a different type.

Here the issue is that the unary `&` operator used in the description requires an lvalue, but the lvalue-to-rvalue conversion is applied to the operand when the target is an rvalue reference type.

It would seem that the lvalue-to-rvalue conversion should not be applied when the target of the cast is an rvalue reference type.

Proposed resolution (July, 2009):

1. Change 8.2.10 [expr.reinterpret.cast] paragraph 1 as follows:

> The result of the expression `reinterpret_cast<T>(v)` is the result of converting the expression `v` to type `T`. If `T` is an lvalue reference type, the result is an lvalue; **if `T` is an rvalue reference type, the result is an rvalue;** otherwise, the result is an rvalue and the lvalue-to-rvalue (7.1 [conv.lval]), array-to-pointer (7.2 [conv.array]), and function-to-pointer (7.3 [conv.func]) standard conversions are performed on the ~~the~~ expression `v`. Conversions that can be performed explicitly using `reinterpret_cast` are listed below. No other conversion can be performed explicitly using `reinterpret_cast`.

2. Change 8.2.11 [expr.const.cast] paragraph 1 as follows:

> The result of the expression `const_cast<T>(v)` is of type `T`. If `T` is an lvalue reference type, the result is an lvalue; **if `T` is an rvalue reference type, the result is an rvalue;** otherwise, the result is an rvalue and the lvalue-to-rvalue (7.1 [conv.lval]), array-to-pointer (7.2 [conv.array]), and function-to-pointer (7.3 [conv.func]) standard conversions are performed on the expression `v`. Conversions that can be performed explicitly using `const_cast` are listed below. No other conversion shall be performed explicitly using `const_cast`.

---

801. Casting away constness in a cast to rvalue reference type

Section: 8.2.11 [expr.const.cast] **Status:** CD2 **Submitter:** UK **Date:** 3 March, 2009

N2800 comment UK 58

[Voted into WP at October, 2009 meeting.]

The rules in 8.2.11 [expr.const.cast] paragraphs 8 and following, defining "casting away constness," do not cover a cast to an rvalue reference type.

Proposed resolution (September, 2009):

Change 8.2.11 [expr.const.cast] paragraph 9 as follows:

> Casting from an lvalue of type `T1` to an lvalue of type `T2` using ~~a~~ **an lvalue** reference cast**, or casting from an expression of type `T1` to an rvalue of type `T2` using an rvalue reference cast,** casts away constness if a cast from an rvalue of type "pointer to `T1`" to the type "pointer to `T2`" casts away constness.

---

891. `const_cast` to rvalue reference from objectless rvalue

Section: 8.2.11 [expr.const.cast] **Status:** CD2 **Submitter:** Steve Adamczyk **Date:** 8 May, 2009

[Voted into WP at March, 2010 meeting.]

8.2.11 [expr.const.cast] paragraph 4 says,

> ...Similarly, for two effective object types `T1` and `T2`, an expression of type `T1` can be explicitly converted to an rvalue of type `T2` using the cast `const_cast<T2&&>` if a pointer to `T1` can be explicitly converted to the type "pointer to `T2`" using a `const_cast`. The result of a reference `const_cast` refers to the original object.

However, in some rvalue-reference `const_cast`s there is no "original object," e.g.,

```
const_cast<int&&>(2)
```

Notes from the July, 2009 meeting:

The coresponding cast to an lvalue reference to const is ill-formed: in such cases, the operand must be an lvalue. The consensus of the CWG was that a cast to an rvalue reference should only accept an rvalue that is an rvalue reference (i.e., an object).

Proposed resolution (February, 2010):

Change 8.2.11 [expr.const.cast] paragraph 4 as follows:

> An lvalue of type `T1` can be explicitly converted to an lvalue of type `T2` using the cast `const_cast<T2&>` (where `T1` and `T2` are object types) if a pointer to `T1` can be explicitly converted to the type "pointer to `T2`" using a `const_cast`. Similarly, for two object types `T1` and `T2`, an ~~expression~~ **lvalue** of type `T1` **or, if `T1` is a class type, an expression of type `T1`,** can be explicitly converted to an rvalue of type `T2` using the cast `const_cast<T2&&>` if a pointer to `T1` can be explicitly converted to the type "pointer to `T2`" using a `const_cast`. The result of a reference `const_cast` refers to the original object.

---

983. Ambiguous pointer-to-member constant

Section: 8.3.1 [expr.unary.op] **Status:** CD2 **Submitter:** Daniel Krügler **Date:** 19 October, 2009

[Voted into WP at March, 2010 meeting.]

The resolution of issue 39 changed the diagnosis of ambiguity because of multiple subobjects from being a lookup error to being diagnosed where the result of the lookup is used. The formation of a pointer to member is one such context but was overlooked in the changes. 8.3.1 [expr.unary.op] paragraph 3 should have language similar to 8.2.5 [expr.ref] paragraph 5 and should be mentioned in the note in 13.2 [class.member.lookup] paragraph 13.

Proposed resolution (October, 2009):

1. Change 8.3.1 [expr.unary.op] paragraph 3 as follows:

> ...For a *qualified-id*, if the member is a static member of type "`T`", the type of the result is plain "pointer to `T`." If the member is a non-static member of class `C` of type `T`, the type of the result is "pointer to member of class `C` of type `T`~~.~~**,**" **and the program is ill-formed if `C` is an ambiguous base (13.2 [class.member.lookup]) of the class designated by the *nested-name-specifier* of the *qualified-id*.**...

2. Change 13.2 [class.member.lookup] paragraph 13 as follows:

> [*Note:* Even if the result of name lookup is unambiguous, use of a name found in multiple subobjects might still be ambiguous (7.12 [conv.mem], 8.2.5 [expr.ref], **8.3.1 [expr.unary.op],** 14.2 [class.access.base]). —*end note*] [*Example:*...

---

803. `sizeof` an enumeration type with a fixed underlying type

Section: 8.3.3 [expr.sizeof] **Status:** CD2 **Submitter:** UK **Date:** 3 March, 2009

N2800 comment UK 70

[Voted into WP at October, 2009 meeting.]

There is no reason for the prohibition of using `sizeof` on "an enumeration type before all its enumerators have been declared" (8.3.3 [expr.sizeof] paragraph 1) if the underlying type of the enumeration is fixed.

Proposed resolution (July, 2009):

Change 8.3.3 [expr.sizeof] paragraph 1 as follows:

> ...The `sizeof` operator shall not be applied to an expression that has function or incomplete type, or to an enumeration type **whose underlying type is not fixed** before all its enumerators have been declared, or to the parenthesized name of such types, or to an lvalue that designates a bit-field...

---

672. Sequencing of initialization in *new-expression*s

Section: 8.3.4 [expr.new] **Status:** CD2 **Submitter:** Clark Nelson **Date:** 11 January, 2008

[Voted into WP at October, 2009 meeting.]

Consider the following code, which uses double-checked locking (DCL):

```
Widget* Widget::Instance() {
  if (pInstance == 0) {          // 1: first check
    lock<mutex> hold(mutW);      // 2: acquire lock
    if (pInstance == 0) {        // 3: second check
      pInstance = new Widget();  // 4: create and assign
    }
  }                              // 5: release lock
}
```

We want this to be fully correct when `pInstance` is an atomic pointer to `Widget`. To get that result, we have to disallow any assignment to `pInstance` until after the new object is fully constructed. In other words, we want this to be an invalid transformation of line 4:

```
pInstance = operator new(sizeof(Widget));
new (pInstance) Widget;
```

I don't think it would be surprising if this were disallowed. For example, if the constructor were to throw an exception, I think many people would expect the variable not to be modified. I think the question is whether it's sufficiently clearly disallowed.

This could be clarified by stating (somewhere appropriate — probably either in 8.3.4 [expr.new] paragraph 16 or paragraph 22) that the initialization of the allocated object is sequenced before the value computation of the *new-expression*. Then by 8.18 [expr.ass] paragraph 1 ("In all cases, the assignment is sequenced after the value computation of the right and left operands, and before the value computation of the assignment expression."), the initialization would have to be sequenced before the assignment.

This is probably not a problem for `atomic<Widget*>` because its `operator=` is a function, and function calls provide the necessary guarantees. But for the plain pointer assignment case, there's still a question about whether the sequencing of side effects is constrained as tightly as it should be. In fact, you don't even have to throw an exception from the constructor for there to be a question.

```
struct X {
    static X* p;
    X();
};

X* X::p = new X;
```

When the constructor for `X` is invoked by this *new-expression*, would it be valid for `X::p` to be non-null? If the answer is supposed to be "no," then I think the Standard should express that intent more clearly.

Proposed resolution (March, 2008):

Change 8.3.4 [expr.new] paragraph 22 as indicated:

> ~~Whether~~ **Initialization of the allocated object is sequenced before the value computation of the *new-expression*. It is unspecified whether** the allocation function is called before evaluating the constructor arguments or after evaluating the constructor arguments but before entering the constructor ~~is unspecified~~. It is also unspecified whether the arguments to a constructor are evaluated if the allocation function returns the null pointer or exits using an exception.

[Drafting note: The editor may wish to move paragraph 22 up to immediately follow paragraph 16 or 17. The paragraphs numbered 18-21 deal with the case where deallocation is done because initialization terminates with an exception, whereas paragraph 22 applies more to the initialization itself, described in paragraph 16.]

Notes from the September, 2008 meeting:

The proposed wording does not (but should) allow the call to the allocation function to occur in the middle of evaluating arguments for the constructor call.

Proposed resolution (July, 2009):

Change 8.3.4 [expr.new] paragraph 21 as follows:

> ~~Whether the allocation function is called before evaluating the constructor arguments or after evaluating the constructor arguments but before entering the constructor is unspecified.~~ **The invocation of the allocation function is indeterminately sequenced with respect to the evaluations of expressions in the *new-initializer*. Initialization of the allocated object is sequenced before the value computation of the *new-expression*.** It is ~~also~~ unspecified whether ~~the arguments to a constructor~~ **expressions in the *new-initializer*** are evaluated if the allocation function returns the null pointer or exits using an exception.

[Drafting note: the editor may wish to consider moving this paragraph to follow paragraph 15 or 16. Paragraphs 17-19 deal with the case where deallocation is done because initialization terminates with an exception, whereas this paragraph applies more to the initialization itself (described in paragraph 15).]

---

804. Deducing the type in `new auto(x)`

Section: 8.3.4 [expr.new] **Status:** CD2 **Submitter:** UK **Date:** 3 March, 2009

N2800 comment UK 71

[Voted into WP at October, 2009 meeting.]

The type of an allocated object wih the type specifier `auto` is determined by the rules of copy initialization, but the initialization applied will be direct initialization. This would affect classes which declare their copy constructor explicit, for instance. For consistency, use the same form of initiailization for the deduction as the new expression.

Proposed resolution (July, 2009):

Change 8.3.4 [expr.new] paragraph 2 as follows:

If the `auto` *type-specifier* appears in the *type-specifier-seq* of a *new-type-id* or *type-id* of a *new-expression*, the *new-expression* shall contain a *new-initializer* of the form

(*assignment-expression*)

The allocated type is deduced from the *new-initializer* as follows: Let ~~`(e)` be~~ `e` **be the *assignment-expression* in** the *new-initializer* and `T` be the *new-type-id* or *type-id* of the *new-expression*, then the allocated type is the type deduced for the variable `x` in the invented declaration (10.1.7.4 [dcl.spec.auto]):

```
T x ~= e~ x(e);
```

[*Example:*...

---

805. Which exception to throw for overflow in array size calculation

Section: 8.3.4 [expr.new] **Status:** CD2 **Submitter:** UK **Date:** 3 March, 2009

N2800 comment UK 72
N2800 comment UK 192

[Voted into WP at July, 2009 meeting as part of N2932.]

Throwing `std::length_error` (8.3.4 [expr.new] paragraph 7) for an attempt to allocate a too-large array brings in too much of the Standard library. A simpler exception, like `std::bad_alloc`, should be thrown instead.

Notes from the March, 2009 meeting:

The CWG was in favor of throwing an exception derived from `std::bad_alloc`. This would be upwardly compatible; it would be harmless for programs that currently catch `std::bad_alloc`, but would allow programs to treat the calculation overflow case separately if they wish.

---

599. Deleting a null function pointer

Section: 8.3.5 [expr.delete] **Status:** CD2 **Submitter:** Martin Sebor **Date:** 3 October 2006

[Voted into WP at July, 2009 meeting.]

The requirements for the operand of the `delete` operators are given in 8.3.5 [expr.delete] paragraph 2:

> In either alternative, the value of the operand of `delete` may be a null pointer value. If it is not a null pointer value, in the first alternative (*delete object*), the value of the operand of `delete` shall be a pointer to a non-array object or a pointer to a subobject (4.5 [intro.object]) representing a base class of such an object (clause 13 [class.derived]). If not, the behavior is undefined. In the second alternative (*delete array*), the value of the operand of `delete` shall be the pointer value which resulted from a previous array *new-expression*. If not, the behavior is undefined.

There are no restrictions on the type of a null pointer, only on a pointer that is not null. That seems wrong.

Proposed resolution (June, 2008):

Change 8.3.5 [expr.delete] paragraph 1 as follows:

> ...The operand shall have a pointer **to object** type, or a class type having a single non-explicit conversion function (15.3.2 [class.conv.fct]) to a pointer **to object** type...

Proposed resolution (September, 2008):

1. Change 8.3.5 [expr.delete] paragraph 1 as follows:

> ...The operand shall have a pointer **to object** type, or a class type having a single non-explicit conversion function (12.3.2) to a pointer **to object** type. **[*Footnote:* This implies that an object cannot be deleted using a pointer of type `void*` because `void` is not an object type. —*end footnote*]** ...

2. Delete the footnote at the end of 8.3.5 [expr.delete] paragraph 3:

> ...if the dynamic type of the object to be deleted differs from its static type, the behavior is undefined. ~~[*Footnote:* This implies that an object cannot be deleted using a pointer of type `void*` because there are no objects of type `void`. —*end footnote*]~~

---

930. `alignof` with incomplete array type

Section: 8.3.6 [expr.alignof] **Status:** CD2 **Submitter:** Alisdair Meredith **Date:** 6 July, 2009

[Voted into WP at October, 2009 meeting.]

8.3.6 [expr.alignof] paragraph 1 currently says regarding `alignof`,

> The operand shall be a *type-id* representing a complete effective object type or a reference to a complete effective object type.

This prohibits taking the alignment of an array type with an unknown bound. There doesn't appear to be any reason for this restriction.

Proposed resolution (July, 2009):

Change 8.3.6 [expr.alignof] paragraph 1 as follows:

> The operand shall be a type-id representing a complete effective object type **or an array thereof** or a reference to a complete effective object type.

---

854. Left shift and unsigned extended types

Section: 8.8 [expr.shift] **Status:** CD2 **Submitter:** Daniel Krügler **Date:** 5 April, 2009

[Voted into WP at October, 2009 meeting.]

According to 8.8 [expr.shift] paragraph 2,

> The value of `E1 << E2` is `E1` (interpreted as a bit pattern) left-shifted `E2` bit positions; vacated bits are zero-filled. If `E1` has an unsigned type, the value of the result is `E1` multiplied by the quantity 2 raised to the power `E2`, reduced modulo `ULLONG_MAX+1` if `E1` has type `unsigned long long int`, `ULONG_MAX+1` if `E1` has type `unsigned long int`, `UINT_MAX+1` otherwise.

This specification does not allow for extended types with rank greater than `long long`; in particular, it says that the value of a shifted unsigned extended type is truncated as if it were the same width as an `unsigned int`.

It's unclear that the second sentence has any normative value; it might be better to relegate it to a note or omit it than to correct it.

Proposed resolution (July, 2009):

Change 8.8 [expr.shift] paragraphs 2-3 as follows:

> The value of `E1 << E2` is `E1` ~~(interpreted as a bit pattern)~~ left-shifted `E2` bit positions; vacated bits are zero-filled. If `E1` has an unsigned type, the value of the result is `E1` ~~multiplied by the quantity 2 raised to the power `E2`~~ **× 2^{E2}**, reduced modulo ~~`ULLONG_MAX+1` if `E1` has type `unsigned long long int`, `ULONG_MAX+1` if `E1` has type `unsigned long int`, `UINT_MAX+1` otherwise. [*Note:* the constants `ULLONG_MAX`, `ULONG_MAX`, and `UINT_MAX` are defined in the header `<climits>`. —*end note*]~~ **one more than the maximum value representable in the result type. Otherwise, if `E1` has a signed type and nonnegative value, and `E1` × 2^{E2} is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.**
>
> The value of `E1 >> E2` is `E1` right-shifted `E2` bit positions. If `E1` has an unsigned type or if `E1` has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of `E1` ~~divided by the quantity 2 raised to the power `E2`~~ **/ 2^{E2}**. If `E1` has a signed type and a negative value, the resulting value is implementation-defined.

---

963. Comparing `nullptr` with 0

Section: 8.9 [expr.rel] **Status:** CD2 **Submitter:** Mike Miller **Date:** 8 September, 2009

[Voted into WP at March, 2010 meeting.]

The current wording of the draft does not indicate what is supposed to happen when an rvalue of type `std::nullptr_t` is compared with an integral null pointer constant. (This could occur, for example, in template code like

```
template<typename T> void f(T t) {
    if (t == 0) // ...
}
```

in a call like `f(nullptr)` -- presumably the body of the template was written before `nullptr` became available and thus used an integral null pointer constant.) Because an integral null pointer constant can be converted to `std::nullptr_t` (7.11 [conv.ptr] paragraph 1), one might expect that 0 would be converted to `std::nullptr_t` and the two operands would compare equal, but 8.9 [expr.rel] paragraph 2 does not handle this case at all, leaving it as undefined behavior.

The current situation is more well-defined (but perhaps not better) with respect to the conditional operator. 8.16 [expr.cond] paragraphs 3-6 make it ill-formed to have `std::nullptr_t` and 0 as the second and third operands. Again, it's not too hard to imagine a legacy function template like

```
template<typename T> void f(T t, bool b) {
    T t = b ? t : 0;
}
```

which would be ill-formed under the current wording of 8.16 [expr.cond].

Either 8.9 [expr.rel] and 8.10 [expr.eq] should be changed to make this combination of operands ill-formed, or those two sections should be changed to give the comparison defined semantics and 8.16 [expr.cond] should be changed to make those operands well-formed.

Proposed resolution (October, 2009):

1. Change 8.9 [expr.rel] paragraph 2 as follows:

   > The usual arithmetic conversions are performed on operands of arithmetic or enumeration type. Pointer conversions (7.11 [conv.ptr]) and qualification conversions (7.5 [conv.qual]) are performed on pointer operands (or on a pointer operand and a null pointer constant**, or on two null pointer constants, at least one of which is non-integral**) to bring them to their composite pointer type. If one operand is a null pointer constant, the composite pointer type is `std::nullptr_t` **if the other operand is also a null pointer constant or, if the other operand is a pointer,** the type of the other operand. Otherwise...

2. Change 8.16 [expr.cond] paragraph 6 bullet 3 as follows:

   - The second and third operands have pointer type, or one has pointer type and the other is a null pointer constant**, or both are null pointer constants, at least one of which is non-integral**; pointer conversions (7.11 [conv.ptr]) and qualification conversions (7.5 [conv.qual]) are performed to bring them to their composite pointer type (8.9 [expr.rel]). The result is of the composite pointer type.

---

587. Lvalue operands of a conditional expression differing only in cv-qualification

Section: 8.16 [expr.cond] **Status:** CD2 **Submitter:** Howard Hinnant **Date:** 20 June 2006

[Voted into WP at October, 2009 meeting.]

Consider the following example:

```
template <typename T>
const T* f(bool b) {
    static T t1 = T();
    static const T t2 = T();
    return &(b ? t1 : t2);  // error?
}
```

According to 8.16 [expr.cond], this function is well-formed if `T` is a class type and ill-formed otherwise. If the second and third operands of a conditional expression are lvalues of the same class type except for cv-qualification, the result of the conditional expression is an lvalue; if they are lvalues of the same non-class type except for cv-qualification, the result is an rvalue.

This difference seems gratuitous and should be removed.

Proposed resolution (June, 2009):

Change 8.16 [expr.cond] paragraph 3 as follows:

> Otherwise, if the second and third operand have different types, and either has (possibly cv-qualified) class type, **or if both are lvalues of the same type except for cv-qualification,** an attempt is made to convert each of those operands to the type of the other. The process for determining whether an operand expression `E1` of type `T1` can be converted to match an operand expression `E2` of type `T2` is defined as follows:
>
> - If `E2` is an lvalue: `E1` can be converted to match `E2` if `E1` can be implicitly converted (Clause 7 [conv]) to the type "lvalue reference to `T2`", subject to the constraint that in the conversion the reference must bind directly (11.6.3 [dcl.init.ref]) to `E1`.
> - If `E2` is an rvalue, or if the conversion above cannot be done **and at least one of the operands has (possibly cv-qualified) class type**:
>   - ...

---

556. Conflicting requirements for acceptable aliasing

Section: 8.18 [expr.ass] **Status:** CD2 **Submitter:** Mike Miller **Date:** 30 January 2006

[Voted into the WP at the March, 2009 meeting.]

There appear to be two different specifications for when aliasing is permitted. One is in 6.10 [basic.lval] paragraph 15:

> If a program attempts to access the stored value of an object through an lvalue of other than one of the following types the behavior is undefined
>
> - the dynamic type of the object,

- a cv-qualified version of the dynamic type of the object,
- a type similar (as defined in 7.5 [conv.qual]) to the dynamic type of the object,
- a type that is the signed or unsigned type corresponding to the dynamic type of the object,
- a type that is the signed or unsigned type corresponding to a cv-qualified version of the dynamic type of the object,
- an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union),
- a type that is a (possibly cv-qualified) base class type of the dynamic type of the object,
- a `char` or `unsigned char` type.

There is also a much more restrictive specification in 8.18 [expr.ass] paragraph 8:

> If the value being stored in an object is accessed from another object that overlaps in any way the storage of the first object, then the overlap shall be exact and the two objects shall have the same type, otherwise the behavior is undefined.

This affects, for example, the definedness of operations on union members: when may a value be stored into one union member and accessed via another.

It should be noted that this conflict existed in C90 and is unchanged in C99 (see, for example, section 6.5 paragraph 7 and section 6.5.16.1 paragraph 3 of ISO/IEC 9899:1999, which directly parallel the sections cited above).

Notes from the October, 2006 meeting:

This issue is based on a misunderstanding of the intent of the wording in 8.18 [expr.ass] paragraph 8. Instead of being a general statement about aliasing, it's describing the situation in which the source of the value being assigned is storage that overlaps the storage of the target object. The proposed resolution should make that clearer rather than changing the specification.

Proposed resolution (June, 2008):

Add the following note at the end of 8.18 [expr.ass] paragraph 8:

> If the value being stored in an object is accessed from another object that overlaps in any way the storage of the first object, then the overlap shall be exact and the two objects shall have the same type, otherwise the behavior is undefined. **[*Note:* This restriction applies to the relationship between the left and right sides of the assignment operation; it is not a statement about how the target of the assignment may be aliased in general. See 6.10 [basic.lval]. —*end note*]**

---

855. Incorrect comments in *braced-init-list* assignment example

Section: 8.18 [expr.ass] **Status:** CD2 **Submitter:** Daniel Krügler **Date:** 5 April, 2009

[Voted into WP at October, 2009 meeting.]

8.18 [expr.ass] paragraph 9 has the following example:

```
complex<double> z;
z = { 1,2 };       // meaning z.operator=(1,2)
z += { 1, 2 };     // meaning z.operator+=(1,2)
```

These comments make it look as if the assignment operator takes two arguments, which is obviously not the case. It would be better if the comments read something like

```
// meaning z.operator=(complex<double>(1,2))
```

or even

```
// meaning z.operator=({1,2}), resolves to
// z.operator=(complex<double>(1,2)
```

Proposed resolution (July, 2009):

Change the example in 8.18 [expr.ass] paragraph 9 as follows:

[*Example:*

```
complex<double> z;
z = { 1,2 };         // meaning z.operator=({1,2})
z += { 1, 2 };       // meaning z.operator+=({1,2})
int a, b;
a = b = { 1 };       // meaning a=b=1;
a = { 1 } = b;       // syntax error
```

—*end example*]

652. Compile-time evaluation of floating-point expressions

Section: 8.20 [expr.const] **Status:** CD2 **Submitter:** Jens Maurer **Date:** 3 October 2007

[Voted into the WP at the March, 2009 meeting.]

It was the intention of the constexpr proposal that implementations be required to evaluate floating-point expressions at compile time. This intention is not reflected in the actual wording of 8.20 [expr.const] paragraph 2, bullet 5:

- a type conversion from a floating-point type to an integral type (7.10 [conv.fpint]) unless the conversion is directly applied to a floating-point literal;

This restriction has the effect of forbidding the use of floating-point expressions in integral constant expressions.

Proposed resolution (June, 2008):

Delete bullet 6 of 8.20 [expr.const] paragraph 2:

- ~~a type conversion from a floating-point type to an integral type (7.10 [conv.fpint]) unless the conversion is directly applied to a floating-point literal;~~

Notes from the June, 2008 meeting:

The CWG agreed with the intent of this issue, that floating-point calculations should be permitted in constant expressions, but acknowledged that this opens the possibility of differing results between compile time and run time. Such issues should be addressed non-normatively, e.g., via a "recommended practice" note like that of C99's 6.4.4.2 or in a technical report.

Proposed resolution (August, 2008):

1. Delete bullet 6 of 8.20 [expr.const] paragraph 2:
   - ~~a type conversion from a floating-point type to an integral type (7.10 [conv.fpint]) unless the conversion is directly applied to a floating-point literal;~~
2. Add a new paragraph after 8.20 [expr.const] paragraph 3:

   [*Note:* Although in some contexts constant expressions must be evaluated during program translation, others may be evaluated during program execution. Since this International Standard imposes no restrictions on the accuracy of floating-point operations, it is unspecified whether the evaluation of a floating-point expression during translation yields the same result as the evaluation of the same expression (or the same operations on the same values) during program execution. [*Footnote:* Nonetheless, implementations are encouraged to provide consistent results, irrespective of whether the evaluation was actually performed during translation or during program execution. —*end footnote*] [*Example:*

   ```
   bool f() {
     char array[1 + int(1 + 0.2 - 0.1 - 0.1)];   // Must be evaluated during translation
     int size = 1 + int(1 + 0.2 - 0.1 - 0.1);    // May be evaluated at runtime
     return sizeof(array) == size;
   }
   ```

   It is unspecified whether the value of `f()` will be `true` or `false`. —*end example*] —*end note*]

---

715. Class member access constant expressions

Section: 8.20 [expr.const] **Status:** CD2 **Submitter:** Steve Adamczyk **Date:** 17 September, 2008

[Voted into WP at October, 2009 meeting.]

Bullet 12 of paragraph 2 of 8.20 [expr.const] says,

- a class member access (8.2.5 [expr.ref]) unless its *postfix-expression* is of effective literal type or of pointer to effective literal type;

This wording needs to be clearer that the "effective literal type" provision applies only to the `.` form of member access and the "pointer to effective literal type" applies only to the `->` form.

Proposed resolution (March, 2009):

Delete 8.20 [expr.const] paragraph 2 bullet 11:

- ~~a class member access (8.2.5 [expr.ref]) unless its postfix-expression is of effective literal type or of pointer to effective literal type;~~

---

721. Where must a variable be initialized to be used in a constant expression?

Section: 8.20 [expr.const] **Status:** CD2 **Submitter:** James Kanze **Date:** 22 September, 2008

[Voted into WP at October, 2009 meeting.]

8.20 [expr.const] paragraph 2 allows an lvalue-to-rvalue conversion in a constant expression if it is applied to "an lvalue of effective integral type that refers to a non-volatile const variable or static data member initialized with constant expressions." However, this does not require, as it presumably should, that the initialization occur in the same translation unit and precede the constant expression, nor that the static data member be initialized within the *member-specification* of its class.

Proposed resolution (March, 2009):

Change 8.20 [expr.const] paragraph 2, bullet 4, sub-bullet 1 as follows:

- an lvalue of effective integral type that refers to a non-volatile const variable **with a preceding initialization** or **to a non-volatile const** static data member **with an initialization in the class definition (12.2.3.2 [class.static.data]), in either case** initialized with constant expressions, or

Additional note, June, 2009:

It has been suggested that the requirement that a static data member be initialized in the class definition is not actually needed but that static data members should be treated like other variable declarations -- a preceding definition with initialization should be sufficient. That is, given

```
extern const int i;
const int i = 5;
struct S {
  static const int j;
};
const int S::j = 5;
int a1[i];
int a2[S::j];
```

there doesn't appear to be a good rationale for making `a1` well-formed and `a2` ill-formed. Some major implementations accept the declaration of `a2` without error.

Proposed resolution (July, 2009):

Change 8.20 [expr.const] paragraph 2, bullet 4, sub-bullet 1 as follows:

- an lvalue of effective integral type that refers to a non-volatile const variable ~~or static data member~~ **with a preceding initialization,** initialized with **a** constant expression~~s~~, or

---

806. Enumeration types in integral constant expressions

Section: 8.20 [expr.const] **Status:** CD2 **Submitter:** UK **Date:** 3 March, 2009

N2800 comment UK 50

[Voted into WP at October, 2009 meeting.]

According to 8.20 [expr.const] paragraph 2, bullet 4, sub-bullet 1, a non-volatile const variable or static data member initialized with constant expressions can be used in an integral constant expression only if it is "of effective integral type." Unscoped enumeration types should also be accepted in such contexts.

Proposed resolution (September, 2009):

Change 8.20 [expr.const] paragraph 2, bullet 4, sub-bullet 1 as indicated:

- an lvalue-to-rvalue conversion (7.1 [conv.lval]) unless it is applied to
  - an lvalue of effective integral **or enumeration** type that refers to a non-volatile const variable or static data member initialized with constant expressions, or
  - ...

1010. Address of object with dynamic storage duration in constant expression

Section: 8.20 [expr.const] **Status:** CD2 **Submitter:** Adamczyk **Date:** 2009-12-02

[Voted into WP at March, 2010 meeting as part of document N3078.]

8.20 [expr.const] paragraph 2 prohibits the unary `&` operator and an array-to-pointer conversion on operands with automatic and thread storage duration, but operands with dynamic storage duration are apparently allowed. Both these operations should be allowed only on operands with static storage duration.

---

569. Spurious semicolons at namespace scope should be allowed

Section: 10 [dcl.dcl] **Status:** CD2 **Submitter:** Matt Austern **Date:** 20 March 2006

[Voted into the WP at the March, 2009 meeting.]

The grammar in 10 [dcl.dcl] paragraph 1 says that a *declaration-seq* is either *declaration* or *declaration-seq declaration*. Some declarations end with semicolons and others (e.g. function definitions and namespace declarations) don't. This means that users who put a semicolon after every declaration are technically writing ill-formed code. The trouble is that in this respect the standard is out of sync with reality. It's convenient to allow semicolons after every declaration, and there's no implementation difficulty in doing so. All existing compilers accept this, except in extra-pedantic mode. When all implementations disagree with the standard, it's time for the standard to change.

Suggested resolution:

In the grammar in 10 [dcl.dcl] paragraph 11, change the second line in the definition of *declaration-seq* to

declaration-seq ; *opt* *declaration*

Proposed resolution (October, 2006):

1. Add the indicated lines to the grammar definitions in 10 [dcl.dcl] paragraph 1:

declaration:
...
namespace-definition
empty-declaration

...

static_assert-declaration:
`static_assert` (*constant-expression* , *string-literal*) ;

empty-declaration:
;

2. Add the following as a new paragraph after 10 [dcl.dcl] paragraph 4:

An *empty-declaration* has no effect.

---

808. Non-type *decl-specifier*s versus max-munch

Section: 10.1 [dcl.spec] **Status:** CD2 **Submitter:** UK **Date:** 3 March, 2009

N2800 comment UK 83

[Voted into WP at March, 2010 meeting.]

According to 10.1 [dcl.spec] paragraph 2,

> The longest sequence of *decl-specifier*s that could possibly be a type name is taken as the *decl-specifier-seq* of a *declaration*.

However, there are many *decl-specifier*s that cannot appear in a type name that are, nonetheless, part of a *declaration*'s *decl-specifier-seq*, such as `typedef`, `friend`, `static`, etc.

Proposed resolution (November, 2009):

Change 10.1 [dcl.spec] paragraph 2 as follows:

> ~~The longest sequence of *decl-specifier*s that could possibly be a type name is taken as the *decl-specifier-seq* of a *declaration*~~ **If a *type-name* is encountered while parsing a *decl-specifier-seq*, it is interpreted as part of the *decl-specifier-seq* if and only if there is no previous *type-specifier* other than a *cv-qualifier* in the *decl-specifier-seq*.**. The sequence shall be self-consistent as described below. [*Example:*...

---

717. Unintentional restrictions on the use of `thread_local`

Section: 10.1.1 [dcl.stc] **Status:** CD2 **Submitter:** Clark Nelson **Date:** 17 September, 2008

N2800 comment US 36

[Voted into WP at October, 2009 meeting.]

The current wording unintentionally restricts the use of the `thread_local` specifier in two contexts: block-scope extern variable declarations and static data members. These restrictions are in conflict with 10.1.1 [dcl.stc] paragraph 1.

Proposed resolution (July, 2009):

Change 10.1.1 [dcl.stc] paragraph 4 as follows:

> The `thread_local` specifier shall be applied only to the names of objects or references of namespace scope ~~and~~**,** to the names of objects or references of block scope that also specify `extern` **or** `static`**, and to the names of static data members**. It specifies that the named object or reference has thread storage duration (6.7.2 [basic.stc.thread]).

---

809. Deprecation of the `register` keyword

Section: 10.1.1 [dcl.stc] **Status:** CD2 **Submitter:** UK **Date:** 3 March, 2009

N2800 comment UK 86

[Voted into WP at October, 2009 meeting.]

The `register` keyword serves very little function, offering no more than a hint that a note says is typically ignored. It should be deprecated in this version of the standard, freeing the reserved name up for use in a future standard, much like `auto` has been re-used this time around for being similarly useless.

Notes from the March, 2009 meeting:

The consensus of the CWG was in favor of deprecating `register`.

Proposed resolution (September, 2009):

1. Change 10.1.1 [dcl.stc] paragraph 3 as follows:

> A `register` specifier is a hint to the implementation that the object so declared will be heavily used. [*Note:* the hint can be ignored and in most implementations it will be ignored if the address of the object is taken. **This use is deprecated (see [depr.register]).** *—end note*]

2. Add a new section following _N3000_.D.4 [depr.string]:

> **register keyword [depr.register]**
>
> **The use of the `register` keyword as a *storage-class-specifier* is deprecated (see 10.1.1 [dcl.stc]).**

---

810. Block-scope `thread_local` variables should be implicitly `static`

Section: 10.1.1 [dcl.stc] **Status:** CD2 **Submitter:** UK **Date:** 3 March, 2009

N2800 comment UK 87

[Voted into WP at March, 2010 meeting.]

According to 10.1.1 [dcl.stc] paragraph 4,

> The `thread_local` specifier shall be applied only to the names of objects or references of namespace scope and to the names of objects or references of block scope that also specify `static`.

Why require two keywords, where one on its own becomes ill-formed? `thread_local` should imply `static` in this case, and the combination of keywords should be banned rather than required. This would also eliminate the one of two exceptions documented in paragraph 1.

Notes from the July, 2009 meeting:

The consensus of the CWG was that `thread_local` should imply `static`, as suggested, but that the combination should still be allowed (it is needed, for example, for thread-local static data members).

Proposed resolution (October, 2009):

Change 10.1.1 [dcl.stc] paragraph 4 as follows:

> The `thread_local` specifier **indicates that the named entity has thread storage duration (6.7.2 [basic.stc.thread]). It** shall be applied only to the names of objects or references of namespace ~~scope, to the names of objects or references of~~ **or** block scope ~~that also specify `extern` or `static`,~~ and to the names of static data members. ~~It specifies that the named object or reference has thread storage duration (6.7.2 [basic.stc.thread]).~~ **When `thread_local` is applied to a variable of block scope the *storage-class-specifier* `static` is implied if it does not appear explicitly.**

---

940. Global anonymous unions

Section: 10.1.1 [dcl.stc] **Status:** CD2 **Submitter:** UK **Date:** 14 July, 2009

N2800 comment UK 85

[Voted into WP at October, 2009 meeting.]

10.1.1 [dcl.stc] paragraph 1 refers to "global anonymous unions." This reference should include anonymous unions declared in a named namespace, not just in global scope (cf 12.3 [class.union] paragraph 3).

Proposed resolution (September, 2009):

Change 10.1.1 [dcl.stc] paragraph 1 as follows:

> If a *storage-class-specifier* appears in a *decl-specifier-seq*, there can be no `typedef` specifier in the same *decl-specifier-seq* and the *init-declarator-list* of the declaration shall not be empty (except for ~~global~~ **an** anonymous union~~s~~ **declared in a named namespace or in the global namespace**, which shall be declared `static` (9.5))...

---

765. Local types in inline functions with external linkage

Section: 10.1.2 [dcl.fct.spec] **Status:** CD2 **Submitter:** Mike Miller **Date:** 6 February, 2009

[Voted into WP at March, 2010 meeting.]

10.1.2 [dcl.fct.spec] paragraph 4 specifies that local static variables and string literals appearing in the body of an inline function with external linkage must be the same entities in every translation unit in the program. Nothing is said, however, about whether local types are likewise required to be the same.

Although a conforming program could always have determined this by use of `typeid`, recent changes to C++ (allowing local types as template type arguments, lambda expression closure classes) make this question more pressing.

Notes from the July, 2009 meeting:

The types are intended to be the same.

Proposed resolution (November, 2009):

Change 10.1.2 [dcl.fct.spec] paragraph 4 as follows:

> ...A `static` local variable in an `extern inline` function always refers to the same object. A string literal in the body of an `extern inline` function is the same object in different translation units. [*Note:* A string literal appearing in a default argument expression is not in the body of an inline function merely because the expression is used in a function call from that inline function. — *end note*] **A type defined within the body of an `extern inline` function is the same type in every translation unit.**

---

576. Typedefs in function definitions

Section: 10.1.3 [dcl.typedef] **Status:** CD2 **Submitter:** Jon Caves **Date:** 21 April 2006

[Voted into the WP at the March, 2009 meeting.]

10.1.3 [dcl.typedef] paragraph 1 says,

> The `typedef` specifier shall not be used in a *function-definition* (11.4 [dcl.fct.def])...

Does this mean that the following is ill-formed?

```
void f() {
    typedef int INT;
}
```

Proposed resolution (March, 2008):

Change 10.1.3 [dcl.typedef] paragraph 1 as follows:

> ...The `typedef` specifier ~~shall not be used in a *function-definition* (11.4 [dcl.fct.def]), and it~~ shall not be combined in a *decl-specifier-seq* with any other kind of specifier except a *type-specifier*, **and it shall not be used in the declaration of a function parameter nor in the *decl-specifier-seq* of a *function-definition* (11.4 [dcl.fct.def])**...

Proposed resolution (September, 2008):

Change 10.1.3 [dcl.typedef] paragraph 1 as follows:

> ...The `typedef` specifier ~~shall not be used in a *function-definition* (11.4 [dcl.fct.def]), and it~~ shall not be combined in a *decl-specifier-seq* with any other kind of specifier except a *type-specifier*, **and it shall be used neither in the *decl-specifier-seq* of a *parameter-declaration* (11.3.5 [dcl.fct]) nor in the *decl-specifier-seq* of a *function-definition* (11.4 [dcl.fct.def])**.

---

699. Must constexpr member functions be defined in the class *member-specification*?

Section: 10.1.5 [dcl.constexpr] **Status:** CD2 **Submitter:** Mike Miller **Date:** 26 June, 2008

N2800 comment UK 49
N2800 comment JP 12
N2800 comment DE 23

[Voted into WP at October, 2009 meeting.]

According to 10.1.5 [dcl.constexpr] paragraph 1,

> The `constexpr` specifier shall be applied only to the definition of an object, function, or function template, or to the declaration of a static data member of a literal type (6.9 [basic.types]).

As a result, a constexpr member function cannot be simply declared in the class *member-specification* and defined later; it must be defined in its initial declaration.

This restriction was not part of the initial proposal but was added during the CWG review. However, the original intent is still visible in some of the wording in 10.1.5 [dcl.constexpr]. For example, paragraph 2 refers to applying the `constexpr` specifier to the "declaration" and not the "definition" of a function or constructor. Although that is formally correct, as definitions are also declarations, it could be confusing. Also, the example in paragraph 6 reads,

```
class debug_flag {
public:
  explicit debug_flag(bool);
  constexpr bool is_on();     // error: debug_flag not literal type
  ...
```

when the proximate error is that `is_on` is only declared, not defined. There are also many occurrences of the `constexpr` specifier in the library clauses where the member function is only declared, not defined.

It's not clear how much simplification is gained by this restriction. There are reasons for defining ordinary inline functions outside the class *member-specification* (reducing the size and complexity of the class definition, separating interface from implementation, making the editing task easier if program evolution results in an inline function being made non-inline, etc.) that would presumably apply to constexpr member functions as well. It seems feasible to allow separate declaration and definition of a constexpr function; it would simply not be permitted to use it in a constant expression before the definition is seen (although it could presumably still be used in non-constant expressions in that region, like an ordinary inline function).

If the prohibition were relaxed to allow separate declaration and definition of constexpr member functions, some questions would need to be answered, such as whether the `constexpr` specifier must appear on both declaration and definition (the `inline` specifier need not). If it can be omitted in one or the other, there's a usability issue regarding the fact that `constexpr` implies `const`; the `const` qualifier would need to be specified explicitly in the declaration in which `constexpr` was omitted.

If the current restriction is kept, the library clauses should state in an introduction that a non-defining declaration of a constexpr member function should be considered "for exposition only" and not literal code.

Notes from the September, 2008 meeting:

In addition to the original issues described above, the question has arisen whether recursive constexpr functions are or should be permitted. Although they were originally desired by the proposers of the feature, they were prohibited out of an abundance of caution. However, the wording that specified the prohibition was changed during the review process, inadvertently permitting them.

The CWG felt that there are sufficient use cases for recursion that it should not be forbidden (although a new minimum for recursion depth should be added to Annex B [implimits]). If mutual recursion is to be allowed, forward declaration of constexpr functions must also be permitted (answering the original question in this issue). A call to an undefined constexpr function in the body of a constexpr function should be diagnosed when the outer constexpr function is invoked in a context requiring a constant expression; in all other contexts, a call to an undefined constexpr function should be treated as a normal runtime function call, just as if it had been invoked with non-constant arguments.

Proposed resolution (July, 2009):

1. Change 8 [expr] paragraph 4 as follows:

> If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined~~, unless such an expression appears where an integral constant expression is required (8.20 [expr.const]), in which case the program is ill-formed~~. [*Note:* most existing implementations of C++ ignore integer overflows. Treatment of division by zero, forming a remainder using a zero divisor, and all floating point exceptions vary among machines, and is usually adjustable by a library function. —*end note*]

2. Add the indicated text to 8.20 [expr.const] paragraph 2:

   - ...
   - an invocation of a function other than a constexpr function or a constexpr constructor [*Note:* overload resolution (16.3 [over.match]) is applied as usual —*end note*];
   - **a direct or indirect invocation of an undefined constexpr function or an undefined constexpr constructor outside the definition of a constexpr function or a constexpr constructor;**
   - **a result that is not mathematically defined or not in the range of representable values for its type;**
   - ...

3. Change 10.1.5 [dcl.constexpr] paragraph 1 as follows:

   The `constexpr` specifier shall be applied only to the definition of an object, **the declaration of a** function~~,~~ or function template, or ~~to~~ the declaration of a static data member of an effective literal type (6.9 [basic.types]). **If any declaration of a function or function template has the** `constexpr` **specifier, then all its declarations shall contain the** `constexpr` **specifier. [*Note:* An explicit specialization can differ from the template declaration with respect to the** `constexpr` **specifier. —*end note*]** [*Note:* function parameters cannot be declared `constexpr`. —*end note*] [*Example:*

```
  constexpr int square(int x);        //OK, declaration
~~constexpr int square(int x) {       //OK~~
~~  return x * x;~~
~~}~~
  constexpr int bufsz = 1024;         // OK, definition
  constexpr struct pixel {            // error: pixel is a type
    int x;
    int y;
    constexpr pixel(int);             // OK, declaration
  };
  constexpr pixel::pixel(int a)
    : x(square(a)), y(square(a)) { }  //OK, definition
  constexpr pixel small(2);           // error: square not defined, so small(2)
                                      // not constant (8.20 [expr.const]), so constexpr not satisfied
  constexpr int square(int x) {       // OK, definition
    return x * x;
  }
  constexpr pixel large(4);           // OK, square defined
  int next(constexpr int x) {         // error, not for parameters
    return x + 1;
  }
  extern constexpr int memsz;         // error: not a definition
```

   —*end example*]

4. Add a new section following 20.5.5.5 [member.functions]:

   Implementations shall provide definitions for any non-defining declarations of `constexpr` **functions and constructors within the associated header files.**

5. Add the following bullet to the list in B [implimits] paragraph 2:

   - ...
   - Nested external specifications [1 024].
   - **Recursive constexpr function invocations [512].**
   - ...

(This resolution also resolves issue 695.)

---

991. Reference parameters of constexpr functions and constructors

Section: 10.1.5 [dcl.constexpr] **Status:** CD2 **Submitter:** Gabriel Dos Reis **Date:** 20 October, 2009

[Voted into WP at March, 2010 meeting as document N3078.]

It would be useful if constexpr functions and constructors could take arguments via reference-to-const parameters. See message 15357.

---

811. Unclear implications of const-qualification

Section: 10.1.7.1 [dcl.type.cv] **Status:** CD2 **Submitter:** UK **Date:** 3 March, 2009

N2800 comment UK 89

[Voted into WP at March, 2010 meeting.]

The normative text in 10.1.7.1 [dcl.type.cv] paragraph 2 reads,

> An object declared in namespace scope with a const-qualified type has internal linkage unless it is explicitly declared `extern` or unless it was previously declared to have external linkage. A variable of non-volatile const-qualified integral or enumeration type initialized by an integral constant expression can be used in integral constant expressions (8.20 [expr.const]).

These two sentences parallel the specifications of 10.1.1 [dcl.stc] paragraph 7 and 8.20 [expr.const]. However, the passages are not identical, leading to questions about whether the meanings are the same.

Proposed resolution (October, 2009):

Change 10.1.7.1 [dcl.type.cv] paragraph 2 as follows:

> ~~An object declared in namespace scope with a const-qualified type has internal linkage unless it is explicitly declared `extern` or unless it was previously declared to have external linkage. A variable of non-volatile const-qualified integral or enumeration type initialized by an integral constant expression can be used in integral constant expressions (8.20 [expr.const]).~~ [*Note:* **Declaring a variable `const` can affect its linkage (10.1.1 [dcl.stc]) and its usability in constant expressions (8.20 [expr.const]). As** ~~as~~ described in 11.6 [dcl.init], the definition of an object or subobject of const-qualified type must specify an initializer or be subject to default-initialization. —*end note*]

---

950. Use of `decltype` as a *class-name*

Section: 10.1.7.2 [dcl.type.simple] **Status:** CD2 **Submitter:** Alisdair Meredith **Date:** 3 August, 2009

[Voted into WP at March, 2010 meeting as document N3049.]

In the current specification, a `decltype` resulting in a class type is not a *class-name*, meaning that it cannot be used as a *base-specifier*. There doesn't seem to be any reason not to allow that, and it would be consistent with the proposed outcome of issue 743.

Proposed resolution (February, 2010):

See paper PL22.16/10-0021 = WG21 N3031.

---

988. Reference-to-reference collapsing with `decltype`

Section: 10.1.7.2 [dcl.type.simple] **Status:** CD2 **Submitter:** Michael Wong **Date:** 19 October, 2009

[Voted into WP at March, 2010 meeting.]

References to references are ill-formed, but special provision is made in cases where this occurs via typedefs or template type parameters. A similar provision is probably needed for types resulting from `decltype`:

```
int x, *p = &x;
decltype(*p) &y = *p;  // reference to reference is ill-formed
```

Proposed resolution (October, 2009):

1. Delete 10.1.3 [dcl.typedef] paragraph 9:

> ~~If a typedef `TD` names a type that is a reference to a type `T`, an attempt to create the type "lvalue reference to *cv* `TD`" creates the type "lvalue reference to `T`," while an attempt to create the type "rvalue reference to *cv* `TD`" creates the type `TD`. [*Example:* ... *end example*]~~

2. Delete 17.3.1 [temp.arg.type] paragraph 4:

> ~~If a *template-argument* for a *template-parameter* `T` names a type that is a reference to a type `A`, an attempt to create the type "lvalue reference to *cv* `T`" creates the type "lvalue reference to `A`," while an attempt to create the type "rvalue reference to *cv* `T`" creates the type `T`. [*Example:* ... *end example*]~~

3. Add the following as a new paragraph at the end of 11.3.2 [dcl.ref]:

> **If a typedef (10.1.3 [dcl.typedef]), a type *template-parameter* (17.3.1 [temp.arg.type]), or a *decltype-specifier* (10.1.7.2 [dcl.type.simple]) denotes a type `TR` that is a reference to a type `T`, an attempt to create the type "lvalue reference to *cv* `TR`" creates the type "lvalue reference to `T`," while an attempt to create the type "rvalue reference to *cv* `TR`" creates the type `TR`. [*Example:***

```
int i;
typedef int& LRI;
typedef int&& RRI;
LRI& r1 = i;                    // r1 has the type int&
const LRI& r2 = i;              // r2 has the type int&
const LRI&& r3 = i;             // r3 has the type int&
RRI& r4 = i;                    // r4 has the type int&
```

```
RRI&& r5 = i;                       // r5 has the type int&&

decltype(r2)& r6 = i;               // r6 has the type int&
decltype(r2)&& r7 = i;              // r7 has the type int&
```

—end example]

---

962. Attributes appertaining to class and enum types

Section: 10.1.7.3 [dcl.type.elab] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 2 September, 2009

[Voted into WP at March, 2010 meeting.]

There is a lack of symmetry in the specification of attributes that apply to class and enum types. For example:

```
class X [[attr]];               // #1
typedef class Y [[attr]] YT;    // #2
```

According to 10.1.7.3 [dcl.type.elab] paragraph 1, #1 associates the `attr` attribute with class `X` for all subsequent references. On the other hand, 11.3 [dcl.meaning] paragraph 5 says that #2 associates the `attr` attribute with the type but not with class `Y`.

Existing implementations (Microsoft, GNU, Sun) with attributes place an attribute that is intended to be associated with a class type between the *class-key* and the class name, and it would be preferable to adopt such an approach instead of the contextual approach in the current formulation.

Proposed resolution (October, 2009):

1. Change 6.3.2 [basic.scope.pdecl] paragraph 6 bullet 1 as follows:

   - for a declaration of the form

     class-key ***attribute-specifier*$_{opt}$** *identifier* ~~*attribute-specifier*$_{opt}$~~ ;

     the *identifier* is declared...

2. Change 6.4.4 [basic.lookup.elab] paragraph 2 as follows:

   ...unless the *elaborated-type-specifier* appears in a declaration with the following form:

   class-key ***attribute-specifier*$_{opt}$** *identifier* ~~*attribute-specifier*$_{opt}$~~ ;

   the *identifier* is looked up... if the *elaborated-type-specifier* appears in a declaration with the form:

   class-key ***attribute-specifier*$_{opt}$** *identifier* ~~*attribute-specifier*$_{opt}$~~ ;

   the *elaborated-type-specifier* is a declaration...

3. In 10.1.7.3 [dcl.type.elab], change the grammar and paragraph 1 as follows:

   elaborated-type-specifier:
   class-key ***attribute-specifier*$_{opt}$** : : $_{opr}$ *nested-name-specifier*$_{opt}$ *identifier*
   ...

   An *attribute-specifier* shall not appear in an *elaborated-type-specifier* unless the latter is the sole constituent of a declaration. If an *elaborated-type-specifier* is the sole constituent of a declaration, the declaration is ill-formed unless it is an explicit specialization (17.8.3 [temp.expl.spec]), an explicit instantiation (17.8.2 [temp.explicit]) or it has one of the following forms:

   class-key ***attribute-specifier*$_{opt}$** *identifier* ~~*attribute-specifier*$_{opt}$~~ ;
   ...

4. Change the grammar in 10.2 [dcl.enum] paragraph 1 as follows:

   enum-head:
   enum-key ***attribute-specifier*$_{opt}$** *identifier*$_{opt}$ ~~*attribute-specifier*$_{opt}$~~ *enum-base*$_{opt}$ *attribute-specifier*$_{opt}$
   enum-key ***attribute-specifier*$_{opt}$** *nested-name-specifier identifier*
   ~~*attribute-specifier*$_{opt}$~~ *enum-base*$_{opt}$ *attribute-specifier*$_{opt}$

   opaque-enum-declaration:
   enum-key ***attribute-specifier*$_{opt}$** *identifier* ~~*attribute-specifier*$_{opt}$~~ *enum-base*$_{opt}$ *attribute-specifier*$_{opt}$

5. Change the grammar in 12 [class] paragraph 1 as follows:

   class-head:
   class-key ***attribute-specifier*$_{opt}$** *identifier*$_{opt}$ ~~*attribute-specifier*$_{opt}$~~ *base-clause*$_{opt}$
   class-key ***attribute-specifier*$_{opt}$** *nested-name-specifier identifier* ~~*attribute-specifier*$_{opt}$~~ *base-clause*$_{opt}$

class-key ***attribute-specifier$_{opt}$*** *nested-name-specifier$_{opt}$ simple-template-id*
~~*attribute-specifier$_{opt}$*~~ *base-clause$_{opt}$*

---

625. Use of `auto` as a *template-argument*

Section: 10.1.7.4 [dcl.spec.auto] **Status:** CD2 **Submitter:** John Spicer **Date:** 9 March 2007

[Voted into WP at March, 2010 meeting.]

The `auto` specifier can be used only in certain contexts, as specified in 10.1.7.4 [dcl.spec.auto] paragraphs 2-3:

> Otherwise (`auto` appearing with no type specifiers other than *cv-qualifier*s), the `auto` *type-specifier* signifies that the type of an object being declared shall be deduced from its initializer. The name of the object being declared shall not appear in the initializer expression.
>
> This use of `auto` is allowed when declaring objects in a block (9.3 [stmt.block]), in namespace scope (6.3.6 [basic.scope.namespace]), and in a *for-init-statement* (9.5.3 [stmt.for]). The *decl-specifier-seq* shall be followed by one or more *init-declarator*s, each of which shall have a non-empty *initializer* of either of the following forms:
>
> `=` *assignment-expression*
> `(` *assignment-expression* `)`

It was intended that `auto` could be used only at the top level of a declaration, but it is not clear whether this wording is sufficient to forbid usage like the following:

```
template <class T> struct A {};
template <class T> void f(A<T> x) {}

void g()
{
    f(A<short>());

    A<auto> x = A<short>();
}
```

Notes from the February, 2008 meeting:

It was agreed that the example should be ill-formed.

Proposed resolution (October, 2009):

Change 10.1.7.4 [dcl.spec.auto] paragraph 3 as follows:

> ...~~The~~ `auto` **shall appear as one of the *decl-specifier*s in the *decl-specifier-seq* and the** *decl-specifier-seq* shall be followed by one or more *init-declarator*s, each of which shall have a non-empty *initializer*.

---

711. `auto` with *braced-init-list*

Section: 10.1.7.4 [dcl.spec.auto] **Status:** CD2 **Submitter:** Jason Merrill **Date:** 27 August, 2008

[Voted into WP at July, 2009 meeting.]

One effect of the initializer-list proposal is that now we allow

```
auto x = { 1, 2, 3 };  // decltype(x) is std::initializer_list<int>
```

but not

```
auto ar[3] = { 1, 2, 3 };  // ill-formed
```

This seems unfortunate, as the code for the first could also support the second. Incidentally, I also failed to update the text in 10.1.7.4 [dcl.spec.auto] paragraph 3 which forbids the use of `auto` with *braced-init-list*s, so technically the first form above is currently ill-formed but has defined semantics.

Bjarne Stroustrup:

Is this the thin edge of a wedge? How about

```
vector<auto> v = { 1, 2, 3 };
```

and

```
template<class T> void f(vector<T>& v);
f({1, 2, 3 });
```

(See also issue 625.)

Proposed resolution (March, 2009):

Change 10.1.7.4 [dcl.spec.auto] paragraph 3 as follows:

> ...The *decl-specifier-seq* shall be followed by one or more *init-declarator*s, each of which shall have a non-empty *initializer*. ~~of either of the following forms:~~
>
> > ~~- *assignment-expression*~~
> > ~~(*assignment-expression*)~~

[Drafting note: This change does not address the original issue of the inability to use `auto` *with an array initializer, only the secondary issue of permitted the braced-init-list. The CWG explicitly decided not to support the array case.]*

---

746. Use of `auto` in *new-expression*s

Section: 10.1.7.4 [dcl.spec.auto] **Status:** CD2 **Submitter:** Jason Merrill **Date:** 18 November, 2008

N2800 comment UK 95

[Voted into WP at July, 2009 meeting.]

In listing the acceptable contexts in which the `auto` specifier may appear, 10.1.7.4 [dcl.spec.auto]) paragraph 4 mentions "the *type-specifier-seq* in a *new-type-id*" but not the *type-id* in the parenthesized form; that is, `new auto (42)` is well-formed but `new (auto) (42)` is not. This seems an unnecessary restriction, as well as contradicting 8.3.4 [expr.new] paragraph 2:

> If the `auto` *type-specifier* appears in the *type-specifier-seq* of a *new-type-id* or *type-id* of a *new-expression*...

(See also issue 496.)

Proposed resolution (March, 2009):

Change 10.1.7.4 [dcl.spec.auto] paragraph 4 as follows:

> The `auto` *type-specifier* can also be used in declaring an object in the *condition* of a selection statement (9.4 [stmt.select]) or an iteration statement (9.5 [stmt.iter]), in the *type-specifier-seq* in ~~a~~ **the** *new-type-id* **or *type-id* of a *new-expression*** (8.3.4 [expr.new]), in a *for-range-declaration*...

---

984. "Deduced type" is unclear in `auto` type deduction

Section: 10.1.7.4 [dcl.spec.auto] **Status:** CD2 **Submitter:** Michael Wong **Date:** 19 October, 2009

[Voted into WP at March, 2010 meeting.]

10.1.7.4 [dcl.spec.auto] paragraph 6 says,

> Once the type of a *declarator-id* has been determined according to 11.3 [dcl.meaning], the type of the declared variable using the *declarator-id* is determined from the type of its initializer using the rules for template argument deduction. Let `T` be the type that has been determined for a variable identifier `d`. Obtain `P` from `T` by replacing the occurrences of `auto` with either a new invented type template parameter `U` or, if the initializer is a *braced-init-list* (11.6.4 [dcl.init.list]), with `std::initializer_list<U>`. The type deduced for the variable `d` is then the deduced type determined using the rules of template argument deduction from a function call (17.9.2.1 [temp.deduct.call]), where `P` is a function template parameter type and the initializer for `d` is the corresponding argument.

The reference to "the deduced type" is unclear; it could be taken as referring either to the template parameter or to the function parameter type. 17.9.2.1 [temp.deduct.call] uses the term "deduced `A`," and that usage should be repeated here.

Proposed resolution (October, 2009):

Change 10.1.7.4 [dcl.spec.auto] paragraph 6 as follows:

> ...The type deduced for the variable `d` is then the deduced ~~type~~ `A` determined using the rules of template argument deduction...

---

628. The values of an enumeration with no enumerator

Section: 10.2 [dcl.enum] **Status:** CD2 **Submitter:** Gennaro Prota **Date:** 15 March 2007

N2800 comment UK 96

[Voted into the WP at the March, 2009 meeting.]

According to 10.2 [dcl.enum] paragraph 6, the underlying type of an enumeration with an empty *enumeration-list* is determined as if the *enumeration-list* contained a single enumerator with value 0. Paragraph 7, which specifies the values of an enumeration and the minimum size of bit-field needed represent those values needs a similar provision for empty *enumeration-list*s.

Proposed resolution (March, 2008):

Add the indicated sentence to the end of 10.2 [dcl.enum] paragraph 5:

> ...It is possible to define an enumeration that has values not defined by any of its enumerators. **If the *enumerator-list* is empty, the values of the enumeration are as if the enumeration had a single enumerator with value 0.**

---

862. Undefined behavior with enumerator value overflow

Section: 10.2 [dcl.enum] **Status:** CD2 **Submitter:** Daniel Krügler **Date:** 7 April, 2009

[Voted into WP at October, 2009 meeting.]

The type of an enumerator that has no initializing value in an enumeration whose underlying type is not fixed is given by the third bullet of 10.2 [dcl.enum] paragraph 5:

> the type of the initializing value is the same as the type of the initializing value of the preceding enumerator unless the incremented value is not representable in that type, in which case the type is an unspecified integral type sufficient to contain the incremented value.

This does not address the case in which there is no such type, meaning that it is apparently undefined behavior. Other cases in which an enumeration value is unrepresentable are made ill-formed (see the preceding paragraph for an enumeration with a fixed underlying type and the following paragraph for the case in which the minimum and maximum values cannot be represented by a single type). It would be better if this case were ill-formed as well, instead of causing undefined behavior.

Proposed resolution (July, 2009):

Change 10.2 [dcl.enum] paragraph 5, bullet 3 as follows:

- Otherwise the type of the initializing value is the same as the type of the initializing value of the preceding enumerator unless the incremented value is not representable in that type, in which case the type is an unspecified integral type suficient to contain the incremented value. **If no such type exists, the program is ill-formed.**

---

812. Duplicate names in inline namespaces

Section: 10.3.1 [namespace.def] **Status:** CD2 **Submitter:** JP **Date:** 3 March, 2009

N2800 comment JP 14

[Voted into WP at March, 2010 meeting.]

It is not clear from the specification in 10.3.1 [namespace.def] paragraph 8 how a declaration in an inline namespace should be handled if the name is the same as one in the containing namespace or in an parallel inline namespace. For example:

```
namespace Q {
  inline namespace V1 {
    int i;
    int j;
  }
  inline namespace V2 {
    int j;
  }
  int i;
}
int Q::i = 1;   // Q::i or Q::V1::i?
int Q::j = 2;   // Q::V1::j or Q::V2::j?
```

Proposed resolution (July, 2009):

This issue is resolved by the resolution of issue 861.

---

919. Contradictions regarding inline namespaces

Section: 10.3.1 [namespace.def] **Status:** CD2 **Submitter:** Michael Wong **Date:** 19 June, 2009

[Voted into WP at March, 2010 meeting as part of document N3079.]

According to 10.3.1 [namespace.def] paragraph 8,

> Members of an inline namespace can be used in most respects as though they were members of the enclosing namespace... Furthermore, each member of the inline namespace can subsequently be explicitly instantiated (17.8.2 [temp.explicit]) or explicitly specialized (17.8.3 [temp.expl.spec]) as though it were a member of the enclosing namespace.

However, that assertion is contradicted for class template specializations by 12 [class] paragraph 11:

> If a *class-head* contains a *nested-name-specifier*, the *class-specifier* shall refer to a class that was previously declared directly in the class or namespace to which the *nested-name-specifier* refers...

It is also contradicted for function template specializations by 6.4.3.2 [namespace.qual] paragraph 6:

> In a declaration for a namespace member in which the *declarator-id* is a *qualified-id*, given that the *qualified-id* for the namespace member has the form
>
> *nested-name-specifier unqualified-id*
>
> the *unqualified-id* shall name a member of the namespace designated by the *nested-name-specifier*.

Proposed resolution (November, 2009):

1. Change 12 [class] paragraph 11 as follows:

> If a *class-head* contains a *nested-name-specifier*, the *class-specifier* shall refer to a class that was previously declared directly in the class or namespace to which the *nested-name-specifier* refers**, or in an element of the inline namespace set (10.3.1 [namespace.def]) of that namespace** (i.e., ~~neither~~ **not merely** inherited ~~nor~~ **or** introduced by a *using-declaration*), and the *class-specifier* shall appear in a namespace enclosing the previous declaration.

2. Change 6.4.3.2 [namespace.qual] paragraph 6 as follows:

> In a declaration for a namespace member in which the *declarator-id* is a *qualified-id*, given that the *qualified-id* for the namespace member has the form
>
> *nested-name-specifier unqualified-id*
>
> the *unqualified-id* shall name a member of the namespace designated by the *nested-name-specifier***, or of an element of the inline namespace (10.3.1 [namespace.def]) of that namespace**. [*Example*:...

(Note: this resolution depends on the resolution for issue 861.)

---

921. Unclear specification of inline namespaces

Section: 10.3.1 [namespace.def] **Status:** CD2 **Submitter:** Michael Wong **Date:** 19 June, 2009

[Voted into WP at October, 2009 meeting.]

According to 10.3.1 [namespace.def] paragraph 8,

> Specifically, the inline namespace and its enclosing namespace are considered to be associated namespaces (6.4.2 [basic.lookup.argdep]) of one another, and a *using-directive* (10.3.4 [namespace.udir]) that names the inline namespace is implicitly inserted into the enclosing namespace.

There are two problems with this sentence. First, the concept of namespaces being associated with each other is undefined; 6.4.2 [basic.lookup.argdep] describes how namespaces are associated with types, not with other namespaces. Second, unlike unnamed namespaces, the location of the implicit *using-directive* is not specified.

Proposed resolution (July, 2009):

Change 10.3.1 [namespace.def] paragraph 8 as follows:

> ...Specifically, the inline namespace and its enclosing namespace are ~~considered to be associated namespaces (6.4.2 [basic.lookup.argdep]) of one another~~ **both added to the set of associated namespaces used in argument-dependent lookup (6.4.2 [basic.lookup.argdep]) whenever one of them is**, and a *using-directive* (10.3.4 [namespace.udir]) that names the inline namespace is implicitly inserted into the enclosing namespace **as for an unnamed namespace (10.3.1.1 [namespace.unnamed])**. Furthermore...

---

926. Inline unnamed namespaces

Section: 10.3.1.1 [namespace.unnamed] **Status:** CD2 **Submitter:** Michael Wong **Date:** 29 June, 2009

[Voted into WP at October, 2009 meeting.]

In 10.3.1 [namespace.def] paragraph 1, an *unnamed-namespace-definition* is defined as

```
inline opt namespace { namespace-body }
```

However, there is no provision for the `inline` keyword in the expansion of unnamed namespaces in 10.3.1.1 [namespace.unnamed] paragraph 1. Strictly interpreted, that would mean that the `inline` qualifier is ignored for unnamed namespaces.

Proposed resolution (September, 2009):

Change 10.3.1.1 [namespace.unnamed] paragraph 1 as follows:

> An *unnamed-namespace-definition* behaves as if it were replaced by
>
> ```
> inline opt namespace unique { /* empty body */ }
> using namespace unique ;
> namespace unique { namespace-body }
> ```
>
> where `inline` **appears if and only if it appears in the *unnamed-namespace-definition*,** all occurrences of ***unique*** in a translation unit are replaced by the same identifier**,** and this identifier differs from all other identifiers in the entire program.[87] [*Example:*...

---

986. Transitivity of *using-directive*s versus qualified lookup

Section: 10.3.4 [namespace.udir] **Status:** CD2 **Submitter:** Michael Wong **Date:** 19 October, 2009

[Voted into WP at March, 2010 meeting.]

According to 10.3.4 [namespace.udir] paragraph 4,

> The *using-directive* is transitive: if a scope contains a *using-directive* that nominates a second namespace that itself contains *using-directive*s, the effect is as if the *using-directive*s from the second namespace also appeared in the first.

This is true only for unqualified lookup; the algorithm in 6.4.3.2 [namespace.qual] paragraph 2 gives different results (the transitive closure terminates when a declaration of the name being looked up is found).

Proposed resolution (October, 2009):

Change 10.3.4 [namespace.udir] paragraph 4 as follows:

~~The~~ **For unqualified lookup (6.4.1 [basic.lookup.unqual]), the** *using-directive* is transitive: if a scope contains a *using-directive* that nominates a second namespace that itself contains *using-directive*s, the effect is as if the *using-directive*s from the second namespace also appeared in the first. **[*Note:* For qualified lookup, see 6.4.3.2 [namespace.qual]. —*end note*]** [*Example:*...

---

564. Agreement of language linkage or *linkage-specification*s?

Section: 10.5 [dcl.link] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 8 March 2006

[Voted into the WP at the March, 2009 meeting.]

The wording of 10.5 [dcl.link] paragraph 5 is suspect:

> If two declarations of the same function or object specify different *linkage-specification*s (that is, the *linkage-specification*s of these declarations specify different *string-literal*s), the program is ill-formed if the declarations appear in the same translation unit, and the one definition rule (3.2) applies if the declarations appear in different translation units.

But what if only one of the declarations has a *linkage-specification*, while the other is left with the default C++ linkage? Shouldn't this restriction be phrased in terms of the functions' or objects' language linkage rather than *linkage-specification*s?

(*Additional note [wmm]:* Is the ODR the proper vehicle for enforcing this requirement? This is dealing with declarations, not necessarily definitions. Shouldn't this say "ill-formed, no diagnostic required" instead of some vague reference to the ODR?)

Proposed resolution (June, 2008):

Change 10.5 [dcl.link] paragraph 5 as follows:

> If two declarations ~~of the same function or object~~ **declare functions with the same name and parameter-type-list (11.3.5 [dcl.fct]) to be members of the same namespace or declare objects with the same name to be members of the same namespace** ~~specify different *linkage-specification*s (that is, the *linkage-specification*s of these declarations specify different *string-literal*s)~~ **and the declarations give the names different language linkages**, the program is ill-formed ~~if the declarations appear in the same translation unit, and the one definition rule (6.2 [basic.def.odr]) applies~~**; no diagnostic is required** if the declarations appear in different translation units.

---

814. Attribute to indicate that a function throws nothing

Section: 10.6 [dcl.attr] **Status:** CD2 **Submitter:** US **Date:** 3 March, 2009

N2800 comment US 40

[Voted into WP at March, 2010 meeting as paper N3050.]

A function with an *exception-specification* of `throw()` must be given a `catch(...)` clause to enforce its contract, i.e., to call `std::unexpected()` if it exits with an exception. It would be useful to have an attribute indicating that the function really does throw nothing and thus that the `catch(...)` clause need not be generated.

(See also issue 830.)

Proposed resolution (September, 2009):

See paper PL22.16/09-0162 = WG21 N2972.

---

951. Problems with *attribute-specifier*s

Section: 10.6 [dcl.attr] **Status:** CD2 **Submitter:** Sean Hunt **Date:** 5 August, 2009

[Voted into WP at March, 2010 meeting as document N3067.]

There are a number of problems with the treatment of attributes in the current draft. One issue is the failure to permit attributes to appear at various points in the grammar at which one might plausibly expect them:

- In a *new-type-id* (8.3.4 [expr.new])
- Preceding the *type-specifier-seq* in a *condition* (9.4 [stmt.select])
- In a *for-init-statement* that is an *expression-statement* (9.5 [stmt.iter])
- Preceding the *type-specifier-seq* in a *for-range-declaration* (9.5 [stmt.iter])
- In a reference *ptr-operator* (11 [dcl.decl])
- Preceding the *type-specifier-seq* in a *type-id* (11.1 [dcl.name])
- Preceding the *decl-specifier-seq* in a *parameter-declaration* (11.3.5 [dcl.fct])
- In a *function-definition* (11.4 [dcl.fct.def]) at any of the three locations where they might be expected:
  - preceding the *decl-specifier-seq*
  - following the parameter list (paragraph 2 repeats the syntax from 11.3.5 [dcl.fct] with the conspicuous omission of the *attribute-specifier*)
  - preceding the *compound-statement* of the *function-body* (this would introduce an ambiguity with the *attribute-specifier* following the parameter list that would need to be addressed)
- Preceding the *decl-specifier-seq* of a *member-declaration* (12.2 [class.mem])
- Preceding the *compound-statement* of a *try-block* or *handler* (18 [except])
- Preceding the *type-specifier-seq* of an *exception-declaration* (18 [except])

Another group of problems is the failure to specify to what a given *attribute-specifier* appertains:

- In a *condition* (9.4 [stmt.select])
- In a *for-range-declaration* (9.5.4 [stmt.ranged])
- In a *parameter-declaration* (11.3.5 [dcl.fct])
- In a *conversion-type-id* (15.3.2 [class.conv.fct])

There is also a problem in the specification of the interpretation of an initial *attribute-specifier*. 11.3 [dcl.meaning] paragraph 5 says,

> In a declaration *attribute-specifier*$_{opt}$ `T` *attribute-specifier*$_{opt}$ `D` where `D` is an unadorned identifier the type of this identifier is "`T`". The first optional *attribute-specifier* appertains to the entity being declared.

This wording only covers the case where the declarator is a simple identifier. It leaves unspecified the meaning of the initial *attribute-specifier* with more complex declarators for pointers, references, functions, and arrays.

Finally, something needs to be said about the case where *attribute-specifier*s occur in both the initial position and following the *declarator-id*: is this permitted, and if so, under what constraints?

(See also issue 968.)

Proposed resolution (February, 2010):

See paper PL22.16/10-0023 = WG21 N3033.

---

970. Consistent use of "appertain" and "apply"

Section: 10.6 [dcl.attr] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 28 September, 2009

[Voted into WP at March, 2010 meeting.]

The terms "appertain" and "apply" are used in different ways in different subsections of 10.6 [dcl.attr]. A thorough editorial sweep of the entire section is needed to regularize their usage.

Proposed resolution (October, 2009):

1. Change 10.6.1 [dcl.attr.grammar] paragraph 4 as follows:

   ...If an *attribute-specifier* that appertains to some entity or statement contains an *attribute* that ~~does not~~ **is not allowed to** apply to that entity or statement, the program is ill-formed...

2. Change 10.6.2 [dcl.align] paragraph 1 as follows:

   ...The attribute ~~can~~ **may** be applied to a variable...

3. Change 10.6.8 [dcl.attr.noreturn] paragraph 1 as follows:

   ...The attribute ~~applies~~ **may be applied** to the *declarator-id* in a function declaration...

4. Change _N3225_.7.6.4 [dcl.attr.final] paragraph 1 as follows:

   ...The attribute ~~applies~~ **may be applied** to class definitions...

5. Change _N3225_.7.6.5 [dcl.attr.override] paragraph 1 as follows:

   ...The attribute ~~applies~~ **may be applied** to virtual member functions...

6. Change _N3225_.7.6.5 [dcl.attr.override] paragraph 3 as follows:

   ...The attribute ~~applies~~ **may be applied** to class members...

7. Change _N3225_.7.6.5 [dcl.attr.override] paragraph 5 as follows:

   ...The attribute ~~applies~~ **may be applied** to a class definition.

8. Change 10.6.3 [dcl.attr.depend] paragraph 1 as follows:

   ...The attribute ~~applies~~ **may be applied** to the *declarator-id* of a *parameter-declaration*... The attribute **may** also ~~applies~~ **be applied** to the declarator-id of a function declaration...

---

815. Parameter pack expansion inside attributes

Section: 10.6.1 [dcl.attr.grammar] **Status:** CD2 **Submitter:** UK **Date:** 3 March, 2009

N2800 comment UK 103

[Voted into WP at July, 2009 meeting as N2933.]

Parameter packs should be expanded inside attributes. For example, it would be useful to specify the alignment of each element in a pack expansion using a parallel pack expansion.

---

957. Alternative tokens and *attribute-token*s

Section: 10.6.1 [dcl.attr.grammar] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 26 August, 2009

N2800 comment FR 12

[Voted into WP at March, 2010 meeting.]

10.6.1 [dcl.attr.grammar] paragraph 3 specifies that keywords can be used as *attribute-token*s. However, the alternative tokens in 5.5 [lex.digraph], such as `bitor` and `compl`, are not keywords. The text should be changed to make the alternative tokens acceptable as

*attribute-token*s as well.

Proposed resolution, October, 2009:

Change 10.6.1 [dcl.attr.grammar] paragraph 3 as follows:

> ...~~A~~ **If a** keyword (5.11 [lex.key]) **or an alternative token (5.5 [lex.digraph]) that satisfies the syntactic requirements of an *identifier* (5.10 [lex.name]) is** contained in an *attribute-token*, **it** is considered an identifier...

---

968. Syntactic ambiguity of the attribute notation

Section: 10.6.1 [dcl.attr.grammar] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 23 September, 2009

[Voted into WP at March, 2010 meeting as document N3063.]

The `[[ ... ]]` notation for attributes was thought to be completely unambiguous. However, it turns out that two `[` characters can be adjacent and not be an attribute-introducer: the first could be the beginning of an array bound or subscript operator and the second could be the beginning of a *lambda-introducer*. This needs to be explored and addressed.

(See also issue 951.)

Proposed resolution (November, 2009):

Add the following paragraph at the end of 11.2 [dcl.ambig.res]:

> **Two consecutive left square bracket tokens shall appear only when introducing an *attribute-specifier*. [*Note:* If two consecutive left square brackets appear where an *attribute-specifier* is not allowed, the program is ill-formed even if the brackets match an alternative grammar production. —*end note*] [*Example:***
>
> ```
> int p[10];
> void f() {
>   int x = 42;
>   int(p[[x]{return x;}()]);   // Error: malformed attribute on a nested
>                               // declarator-id and not a function-style cast of
>                               // an element of p.
>   new int[[]{return x;}()];   // Error even though attributes are not allowed
> }                             // in this context.
> ```
>
> *—end example*]

---

959. Alignment attribute for class and enumeration types

Section: 10.6.2 [dcl.align] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 31 August, 2009

[Voted into WP at March, 2010 meeting.]

According to 10.6.2 [dcl.align] paragraph 1, an alignment attribute can be specified only for a variable or a class data member. The corresponding Microsoft and GNU attributes can be also specified for a class type, and this usage seems to be widespread. It should be permitted with the standard attribute and there seems no good reason not to do so for enumeration types, as well.

Notes from the October, 2009 meeting:

Although there was initial concern for how to integrate the suggested change into the type system, it was observed that current practice is to have the attribute affect only the layout, not the type.

Proposed resolution (February, 2010):

1. Change 10.6.2 [dcl.align] paragraphs 1-2 as follows:

> ...The attribute can be applied to a variable that is neither a function parameter nor declared with the register storage class specifier and to a class data member that is not a bit-field. **The attribute can also be applied to the declaration of a class or enumeration type.**
>
> When the alignment attribute is of the form `align(`*assignment-expression*`)`:
>
> - ...
> - if the constant expression evaluates to a fundamental alignment, the alignment requirement of the declared ~~object~~ **entity** shall be the specified fundamental alignment
> - if the constant expression evaluates to an extended alignment and the implementation supports that alignment in the context of the declaration, the alignment of the declared ~~object~~ **entity** shall be that alignment
> - ...

2. Change 10.6.2 [dcl.align] paragraphs 4-6 as follows:

> When multiple alignment attributes are specified for an ~~object~~ **entity**, the alignment requirement shall be set to the strictest specified alignment.
>
> The combined effect of all alignment attributes in a declaration shall not specify an alignment that is less strict than the alignment that would otherwise be required for the ~~object~~ **entity** being declared.
>
> If the defining declaration of an ~~object~~ **entity** has an alignment attribute, any non-defining declaration of that ~~object~~ **entity** shall either specify equivalent alignment or have no alignment attribute. **Conversely, if any declaration of an entity has an alignment attribute, every defining declaration of that entity shall specify an equivalent alignment.** No diagnostic is required if declarations of an ~~object~~ **entity** have different alignment attributes in different translation units.

3. Insert the following as a new paragraph after 10.6.2 [dcl.align] paragraph 6:

> **[*Example:***
>
> ```
> // Translation unit #1:
> struct S { int x; } s, p = &s;
>
> // Translation unit #2:
> struct [[align(16)]] S;    // error: definition of S lacks alignment; no
> extern S* p;               //        diagnostic required
> ```
>
> ***—end example*]**

4. Delete 10.6.2 [dcl.align] paragraph 8:

> ~~[*Note:* the alignment of a union type can be strengthened by applying the alignment attribute to any non-static data member of the union. *—end note*]~~

---

965. Limiting the applicability of the `carries_dependency` attribute

Section: 10.6.3 [dcl.attr.depend] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 15 September, 2009

[Voted into WP at March, 2010 meeting.]

The current wording for the `carries_dependency` attribute does not limit it to value-returning functions (when applied to the *declarator-id*, indicating that the return value is affected), nor does it prohibit use in the declaration of a typedef or function pointer. Arguably these meaningless declarations should be prohibited.

Proposed resolution (October, 2009):

Change 10.6.3 [dcl.attr.depend] paragraph 1 as follows:

> ...The attribute applies to the *declarator-id* of a *parameter-declaration* **in a function declaration or lambda**, in which case it specifies that the initialization of the parameter carries a dependency to (4.7 [intro.multithread]) each lvalue-to-rvalue conversion (7.1 [conv.lval]) of that object. The attribute also applies to the *declarator-id* of a function declaration, in which case it specifies that the return value**, if any,** carries a dependency to the evaluation of the function call expression.

---

770. Ambiguity in late-specified return type

Section: 11 [dcl.decl] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 9 February, 2009

[Voted into the WP at the July, 2009 meeting as part of N2927.]

It is currently unspecified whether a declaration like

```
f() -> struct S { };
```

should be parsed as a declaration of `f` whose return type is a class definition (which will be ill-formed according to 10.1.7 [dcl.type] paragraph 3) or as a definition of `f` whose return type is an *elaborated-type-specifier*.

Proposed resolution (June, 2009):

See document PL22.16/09-0117 = WG21 N2927.

---

979. Position of *attribute-specifier* in declarator syntax

Section: 11 [dcl.decl] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 8 October, 2009

In function, pointer, and pointer-to-member declarators, the *attribute-specifier* appertains to the type being declared, but the syntax has the *attribute-specifier*$_{opt}$ appearing before the full type is seen — i.e., before the *cv-qualifier-seq*$_{opt}$ and, for the function case, before the *ref-qualifier*$_{opt}$. GNU attributes appear after these elements (and, for the function case, after the *exception-specification*$_{opt}$ as well). It would be better, both logically and for consistency with existing practice, to move the *attribute-specifier*$_{opt}$ accordingly.

---

374. Can explicit specialization outside namespace use qualified name?

Section: 11.3 [dcl.meaning] **Status:** CD2 **Submitter:** Steve Adamczyk **Date:** 23 August 2002

[Voted into WP at March, 2010 meeting as document N3064.]

This case is nonstandard by 11.3 [dcl.meaning] paragraph 1 (there is a requirement that the specialization first be declared within the namespace before being defined outside of the namespace), but probably should be allowed:

```
namespace NS1 {
  template<class T>
  class CDoor {
  public:
    int mtd() { return 1; }
  };
}
template<> int NS1::CDoor<char>::mtd()
{
  return 0;
}
```

Notes from October 2002 meeting:

There was agreement that we wanted to allow this.

Proposed resolution (February, 2010):

1. Change 11.3 [dcl.meaning] as follows:

   ...A *declarator-id* shall not be qualified except for the definition of a member function (12.2.1 [class.mfct]) or static data member (12.2.3 [class.static]) outside of its class, the definition or explicit instantiation of a function or variable member of a namespace outside of its namespace, or the definition of ~~a previously declared~~ **an** explicit specialization outside of its namespace, or the declaration of a friend function that is a member of another class or namespace (14.3 [class.friend]). When the *declarator-id* is qualified, the declaration shall refer to a previously declared member of the class or namespace to which the qualifier refers (or of an inline namespace within that scope (10.3.1 [namespace.def])) **or to a specialization thereof**, and the member shall not have been introduced by a *using-declaration* in the scope of the class or namespace nominated by the *nested-name-specifier* of the *declarator-id*. [*Note:*...

2. Change 17.8.3 [temp.expl.spec] paragraphs 2-4 as follows:

   An explicit specialization shall appear in namespace scope. An explicit specialization **whose *declarator-id* is not qualified** shall be declared in the nearest enclosing namespace of the template, or, if the namespace is inline (10.3.1 [namespace.def]), any namespace from its enclosing namespace set. Such a declaration may also be a definition. ~~If the declaration is not a definition, the specialization may be defined later (10.3.1.2 [namespace.memdef]).~~

   A declaration of a function template or class template being explicitly specialized shall ~~be in scope at the point of~~ **precede the** declaration of ~~an~~ **the** explicit specialization. [*Note:* a declaration, but not a definition of the template is required. — *end note*] The definition of a class or class template shall ~~be in scope at the point of~~ **precede the** declaration of an explicit specialization for a member template of the class or class template. [*Example:* ... — *end example*]

   A member function, a member class or a static data member of a class template may be explicitly specialized for a class specialization that is implicitly instantiated; in this case, the definition of the class template shall ~~be in scope at the point of declaration of~~ **preced** the explicit specialization for the member of the class template. If such an explicit specialization for the member of a class template names an implicitly-declared special member function (Clause 15 [special]), the program is ill-formed.

---

920. Interaction of inline namespaces and *using-declaration*s

Section: 11.3 [dcl.meaning] **Status:** CD2 **Submitter:** Michael Wong **Date:** 19 June, 2009

[Voted into WP at March, 2010 meeting as part of document N3079.]

According to 11.3 [dcl.meaning] paragraph 1,

> When the *declarator-id* is qualified, the declaration shall refer to a previously declared member of the class or namespace to which the qualifier refers (or of an inline namespace within that scope (10.3.1 [namespace.def])), and the member shall not have been introduced by a *using-declaration* in the scope of the class or namespace nominated by the *nested-name-specifier* of the *declarator-id*.

This would appear to make the following example ill-formed, even though it would be well-formed if the *using-declaration* were omitted:

```
namespace A {
  inline namespace B {
    template <class T> void foo() { }
  }
  using B::foo;
}
template void A::foo<int>();
```

This seems strange.

Proposed resolution (July, 2009):

Change 11.3 [dcl.meaning] paragraph 1 as follows:

> ...When the *declarator-id* is qualified, the declaration shall refer to a previously declared member of the class or namespace to which the qualifier refers (or**, in the case of a namespace,** of an **element of the** inline namespace ~~within that scope~~ **set of that namespace** (10.3.1 [namespace.def]))~~, and~~**;** the member shall not **merely** have been introduced by a *using-declaration* in the scope of the class or namespace nominated by the *nested-name-specifier* of the *declarator-id*. [*Note:*...

(Note: this resolution depends on the resolution of issue 861.)

---

701. When is the array-to-pointer conversion applied?

Section: 11.3.4 [dcl.array] **Status:** CD2 **Submitter:** Eelis van der Weegen **Date:** 13 July, 2008

[Voted into WP at March, 2010 meeting.]

Paragraph 7 of 11.3.4 [dcl.array] says,

> If `E` is an *n*-dimensional array of rank *i* × *j* × ... × *k*, then `E` appearing in an expression is converted to a pointer to an (*n* - 1)-dimensional array with rank *j* × ... × *k*.

This formulation does not allow for the existence of expressions in which the array-to-pointer conversion does *not* occur (as specified in clause 8 [expr] paragraph 9). This paragraph should be no more than a note, if it appears at all, and the wording should be corrected.

Proposed resolution (November, 2009):

Change paragraphs 6-8 of 11.3.4 [dcl.array] into a note and make the indicated changes:

> **[*Note:*** Except where it has been declared for a class (16.5.5 [over.sub]), the subscript operator `[]` is interpreted in such a way that `E1[E2]` is identical to `*((E1)+(E2))`. Because of the conversion rules that apply to `+`, if `E1` is an array and `E2` an integer, then `E1[E2]` refers to the `E2`-th member of `E1`. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.
>
> A consistent rule is followed for multidimensional arrays. If `E` is an *n*-dimensional array of rank *i* × *j* × . . . × *k*, then `E` appearing in an expression **that is subject to the array-to-pointer conversion (7.2 [conv.array])** is converted to a pointer to an (*n*-1)-dimensional array with rank *j* × . . . × *k*. If the `*` operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to (*n*-1)-dimensional array, which itself is immediately converted into a pointer.
>
> [*Example:* consider
>
> ```
> int x[3][5];
> ```
>
> Here `x` is a 3 × 5 array of integers. When `x` appears in an expression, it is converted to a pointer to (the first of three) five-membered arrays of integers. In the expression `x[i]` which is equivalent to `*(x+i)`, `x` is first converted to a pointer as described; then `x+i` is converted to the type of `x`, which involves multiplying `i` by the length of the object to which the pointer points, namely five integer objects. The results are added and indirection applied to yield an array (of five integers), which in turn is converted to a pointer to the first of the integers. If there is another subscript the same argument applies again; this time the result is an integer. —*end example*] ***—end note*]**

---

713. Unclear note about cv-qualified function types

Section: 11.3.5 [dcl.fct] **Status:** CD2 **Submitter:** Doug Gregor **Date:** 11 September, 2008

[Voted into WP at October, 2009 meeting.]

7.5 [conv.qual] paragraph 3 consists of a note reading,

> [*Note:* Function types (including those used in pointer to member function types) are never cv-qualified (11.3.5 [dcl.fct]). —*end note*]

However, 11.3.5 [dcl.fct] paragraph 7 says,

> A *cv-qualifier-seq* shall only be part of the function type...

This sounds like a contradiction, although formally it is not: a "function type with a *cv-qualifier-seq*" is not a "cv-qualified function type." It would be helpful to make this distinction clearer.

Proposed resolution (March, 2009):

1. Change 11.3.5 [dcl.fct] paragraph 7 as follows:

> A *cv-qualifier-seq* shall only be part of the function type for a non-static member function, the function type to which a pointer to member refers, or the top-level function type of a function typedef declaration. **[*Note:* A function type that has a *cv-qualifier-seq* is not a cv-qualified type; there are no cv-qualified function types. —*end note*]** The effect of a *cv-qualifier-seq* in a function declarator...

2. Change 6.9.3 [basic.type.qualifier] paragraph 3 as follows:

> ...See 11.3.5 [dcl.fct] and 12.2.2.1 [class.this] regarding ~~cv-qualified~~ function types **that have *cv-qualifiers***.

---

818. Function parameter packs in non-final positions

Section: 11.3.5 [dcl.fct] **Status:** CD2 **Submitter:** US **Date:** 3 March, 2009

N2800 comment US 45

[Voted into WP at March, 2010 meeting as part of document N3079.]

11.3.5 [dcl.fct] paragraph 13 requires that a parameter pack, if present, must appear at the end of the parameter list. This restriction is not necessary when template argument deduction is not needed and is inconsistent with the way pack expansions are handled. It should be removed.

(See also issue 692.)

---

956. Function prototype scope with late-specified return types

Section: 11.3.5 [dcl.fct] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 21 August, 2009

[Voted into WP at March, 2010 meeting.]

According to 6.3.4 [basic.scope.proto] paragraph 1,

> In a function declaration, or in any function declarator except the declarator of a function definition (11.4 [dcl.fct.def]), names of parameters (if supplied) have function prototype scope, which terminates at the end of the nearest enclosing function declarator.

Happily, this permits the use of parameter names with `decltype` in a late-specified return type, because the return type is part of the function's declarator. However, the note in 11.3.5 [dcl.fct] paragraph 11 is now inaccurate and should be updated:

> [*Note:* ...If a parameter name is present in a function declaration that is not a definition, it cannot be used outside of the *parameter-declaration-clause* since it goes out of scope at the end of the function declarator (6.3 [basic.scope]). —*end note*]

Proposed resolution (February, 2010):

Change the note in 11.3.5 [dcl.fct] paragraph 10 as follows:

> ...[*Note:* in particular, parameter names are also optional in function definitions and names used for a parameter in different declarations and the definition of a function need not be the same. If a parameter name is present in a function declaration that is not a definition, it cannot be used outside of ~~the *parameter-declaration-clause* since it goes out of scope at the end of the function declarator (6.3 [basic.scope])~~ **its function declarator because that is the extent of its potential scope (6.3.4 [basic.scope.proto])**. —*end note*]

---

777. Default arguments and parameter packs

Section: 11.3.6 [dcl.fct.default] **Status:** CD2 **Submitter:** Michael Wong **Date:** 13 February, 2009

[Voted into WP at March, 2010 meeting.]

11.3.6 [dcl.fct.default] paragraph 4 says,

> In a given function declaration, all parameters subsequent to a parameter with a default argument shall have default arguments supplied in this or previous declarations.

It is not clear whether this applies to parameter packs or not. For example, is the following well-formed?

```
template <typename... T> void f(int i = 0, T ...args) { }
```

Note for comparison the corresponding wording in 17.1 [temp.param] paragraph 11 regarding template parameter packs:

> If a *template-parameter* of a class template has a default *template-argument*, each subsequent *template-parameter* shall either have a default *template-argument* supplied or be a template parameter pack.

Proposed resolution (October, 2009):

Change 11.3.6 [dcl.fct.default] paragraph 4:

> ...In a given function declaration, ~~all~~ **each** parameter~~s~~ subsequent to a parameter with a default argument shall have **a** default argument~~s~~ supplied in this or **a** previous declaration~~s~~ **or shall be a function parameter pack**. A default argument...

---

732. Late-specified return types in function definitions

Section: 11.4 [dcl.fct.def] **Status:** CD2 **Submitter:** Daniel Krügler **Date:** 7 October, 2008

N2800 comment DE 13

[Voted into the WP at the July, 2009 meeting as part of N2927.]

The grammar in 11.4 [dcl.fct.def] paragraph 2 incorrectly excludes late-specified return types and should be corrected.

Proposed resolution (July, 2009)

See document PL22.16/09-0117 = WG21 N2927.

---

845. What is the "first declaration" of an explicit specialization?

Section: 11.4 [dcl.fct.def] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 20 March, 2009

[Voted into WP at March, 2010 meeting.]

According to 11.4 [dcl.fct.def] paragraph 10,

> A deleted definition of a function shall be the first declaration of the function.

The Standard is not currently clear about what the "first declaration" of an explicit specialization of a function template is. For example,

```
template<typename T> void f() { }
template<> void f<int>() = delete;  // First declaration?
```

Proposed resolution (October, 2009):

> A deleted definition of a function shall be the first declaration of the function **or, for an explicit specialization of a function template, the first declaration of that specialization**.

(This resolution also resolves issue 915.)

Notes from the October, 2009 meeting:

It was observed that this specification is complicated by the fact that the "first declaration" of a function might be in a block-extern declaration.

---

906. Which special member functions can be defaulted?

Section: 11.4 [dcl.fct.def] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 27 May, 2009

[Voted into WP at March, 2010 meeting.]

The only restriction placed on the use of "`=default`" in 11.4 [dcl.fct.def] paragraph 9 is that a defaulted function must be a special member function. However, there are many variations of declarations of special member functions, and it's not clear which of those should be able to be defaulted. Among the possibilities:

- default arguments
- by-value parameter for a copy assignment operator

- exception specifications
- arbitrary return values for copy assignment operators
- a `const` reference parameter when the implicit function would have a non-const

Presumably, you should only be able to default a function if it is declared compatibly with the implicit declaration that would have been generated.

Proposed resolution (October, 2009):

1. Change 11.4 [dcl.fct.def] paragraph 9 as follows:

   A function definition of the form:

   decl-specifier-seq$_{opt}$ *attribute-specifier*$_{opt}$ *declarator* `= default ;`

   is called an *explicitly-defaulted* definition. ~~Only special member functions may be explicitly defaulted, and the implementation shall define them as if they had implicit definitions (15.1 [class.ctor], 15.4 [class.dtor], 15.8 [class.copy]).~~ **A function that is explicitly defaulted shall**

   - **be a special member function,**
   - **have the same declared function type (except for possibly-differing *ref-qualifier*s and except that in the case of a copy constructor or copy assignment operator, the parameter type may be "reference to non-const `T`," where `T` is the name of the member function's class) as if it had been implicitly declared,**
   - **not have default arguments, and**
   - **not have an *exception-specification*.**

   [*Note:* This implies that parameter types, return type, and cv-qualifiers must match the hypothetical implicit declaration. —*end note*] An explicitly-defaulted function may be declared constexpr only if it would have been implicitly declared as constexpr. If it is explicitly defaulted on its first declaration,

   - **it shall be public,**
   - **it shall not be explicit,**
   - **it shall not be virtual,**
   - **it is implicitly considered to have the same *exception-specification* as if it had been implicitly declared (18.4 [except.spec]), and**
   - **in the case of a copy constructor or copy assignment operator, it shall have the same parameter type as if it had been implicitly declared.**

   [*Note:* Such a special member function may be trivial, and thus its accessibility and explicitness should match the hypothetical implicit definition; see below. —*end note*] [*Example:*

```
struct S {
  S(int a = 0) = default;              // ill-formed: default argument
  void operator=(const S&) = default;  // ill-formed: non-matching return type
  ~S() throw() = default;              // ill-formed: exception-specification
private:
  S(S&);                               // OK: private copy constructor
};
S::S(S&) = default;                    // OK: defines copy constructor
```

   —*end example*] Explicitly-defaulted functions and implicitly-declared functions are collectively called *defaulted* functions, and the implementation shall provide implicit definitions for them (15.1 [class.ctor], 15.4 [class.dtor], 15.8 [class.copy]), which might mean defining them as deleted. A special member function that would be implicitly defined as deleted may be explicitly defaulted only on its first declaration, in which case it is defined as deleted. A special member function is *user-provided* if it is user-declared and not explicitly defaulted on its first declaration. A user-provided explicitly-defaulted function is defined at the point where it is explicitly defaulted. [*Note:*...

   [Editorial note: this change incorporates the overlapping portion of the resolution of issue 667.]

2. Change 15.1 [class.ctor] paragraph 6 as follows:

   ...[*Note:* ...An explicitly-defaulted definition ~~has no~~ **might have an** implicit *exception-specification***, see 11.4 [dcl.fct.def]**. —*end note*]

This resolution also resolves issue 905. See also issue 667.

---

908. Deleted global allocation and deallocation functions

Section: 11.4 [dcl.fct.def] **Status:** CD2 **Submitter:** John Spicer **Date:** 2 June, 2009

[Voted into WP at October, 2009 meeting.]

According to 11.4 [dcl.fct.def] paragraph 10, a deleted definition of a function must be its first declaration. It is not clear whether this requirement can be satisfied for the global allocation and deallocation functions. According to 6.7.4 [basic.stc.dynamic] paragraph 2, they are "implicitly declared in global scope in each translation unit of a program." However, that does not specify where in the translation unit the declaration is considered to take place. This needs to be clarified.

Proposed resolution (July, 2009):

Change 11.4 [dcl.fct.def] paragraph 10 as follows:

> ...A deleted definition of a function shall be the first declaration of the function. **An implicitly declared allocation or deallocation function (6.7.4 [basic.stc.dynamic]) shall not be defined as deleted.** [*Example:*...

---

915. Deleted specializations of member function templates

Section: 11.4 [dcl.fct.def] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 12 June, 2009

[Voted into WP at March, 2010 meeting.]

It is not clear whether the following definition of an explicit specialization of a member function template is permitted or not:

```
template <typenanme T> struct S {
  template <typename U> void f();
};
template <> template <typename U>
  void S<int>::f() = delete;
```

Is the explicit specialization the "first declaration" of the member function template?

(See also issue 845.)

Notes from the July, 2009 meeting:

The intent is that this usage should be supported.

Proposed resolution (October, 2009):

This issue is resolved by the resolution of issue 845.

---

928. Defaulting a function that would be implicitly defined as deleted

Section: 11.4 [dcl.fct.def] **Status:** CD2 **Submitter:** Alisdair Meredith **Date:** 1 July, 2009

[Voted into WP at October, 2009 meeting.]

11.4 [dcl.fct.def] paragraph 9 says,

> A special member function that would be implicitly defined as deleted shall not be explicitly defaulted.

It would be more regular (and thus useful in generic programming) if such a member function were itself simply defined as deleted rather than being made ill-formed.

Proposed resolution (July, 2009):

1. Change 11.4 [dcl.fct.def] paragraph 9 as follows:

   > Only special member functions may be explicitly defaulted, and the implementation shall define them as if they had implicit definitions (15.1 [class.ctor], 15.4 [class.dtor], 15.8 [class.copy]). ~~A special member function that would be implicitly defined as deleted shall not be explicitly defaulted.~~ **A special member function that would be implicitly defined as deleted may be explicitly defaulted only on its first declaration, in which case it is defined as deleted.** A special member function is user-provided if...

2. Change 15.1 [class.ctor] paragraph 6 as follows:

   > A non-user-provided default constructor for a class is implicitly defined when it is used (6.2 [basic.def.odr]) to create an object of its class type (4.5 [intro.object]). ~~If the implicitly-defined default constructor is explicitly defaulted but the corresponding implicit declaration would have been deleted, the program is ill-formed.~~ The implicitly-defined or explicitly-defaulted default constructor...

3. Change 15.4 [class.dtor] paragraph 4 as follows:

   > A program is ill-formed if the class for which a destructor is implicitly defined or explicitly defaulted has: ~~if the implicitly-defined destructor is explicitly-defaulted, but the corresponding implicit declaration would have been deleted.~~
   >
   > - a non-static data member of class type (or array thereof) with an inaccessible destructor, or

- a base class with an inaccessible destructor.

4. Change 15.8 [class.copy] paragraph 7 as follows:

> ...[*Note:* the copy constructor is implicitly defined even if the implementation elided its use (15.2 [class.temporary]). —*end note*] ~~A program is ill-formed if the implicitly-defined copy constructor is explicitly defaulted, but the corresponding implicit declaration would have been deleted.~~

5. Change 15.8 [class.copy] paragraph 12 as follows:

> A non-user-provided copy assignment operator is *implicitly defined* when an object of its class type is assigned a value of its class type or a value of a class type derived from its class type. ~~A program is ill-formed if the implicitly-defined copy assignment operator is explicitly defaulted, but the corresponding implicit declaration would have been deleted.~~

---

611. Zero-initializing references

Section: 11.6 [dcl.init] **Status:** CD2 **Submitter:** Alisdair Meredith **Date:** 29 December 2006

According to 11.6 [dcl.init] paragraph 5,

> To *zero-initialize* an object of type `T` means:
>
> - ...
> - if `T` is a reference type, no initialization is performed.

However, a reference is not an object, so this makes no sense.

Proposed resolution (March, 2010):

This issue is resolved by the resolution of issue 633 in document N2993.

---

869. Uninitialized `thread_local` objects

Section: 11.6 [dcl.init] **Status:** CD2 **Submitter:** Daniel Krügler **Date:** 14 April, 2009

[Voted into WP at March, 2010 meeting.]

11.6 [dcl.init] paragraph 11 says,

> If no initializer is specified for an object, the object is default-initialized; if no initialization is performed, a non-static object has indeterminate value.

This is inaccurate, because objects with thread storage duration are zero-initialized (6.6.2 [basic.start.static] paragraph 2).

Proposed resolution (November, 2009):

Change 11.6 [dcl.init] paragraph 11 as follows:

> If no initializer is specified for an object, the object is default-initialized; if no initialization is performed, ~~a non-static~~ **an** object **with automatic or dynamic storage duration** has indeterminate value. [*Note:* objects with static **or thread** storage duration are zero-initialized, see 6.6.2 [basic.start.static]**.** —*end note*]~~.~~

---

886. Member initializers and aggregates

Section: 11.6.1 [dcl.init.aggr] **Status:** CD2 **Submitter:** Daniel Krügler **Date:** 5 May, 2009

[Voted into WP at March, 2010 meeting.]

The current wording of 11.6.1 [dcl.init.aggr] paragraph 1 does not consider *brace-or-equal-initializer*s on members as affecting whether a class type is an aggregate or not. Because in-class member initializers are essentially syntactic sugar for *mem-initializer*s, and the presence of a user-provided constructor disqualifies a class from being an aggregate, presumably the same should hold true of member initializers.

Proposed resolution (November, 2009):

Change 11.6.1 [dcl.init.aggr] paragraph 1 as follows:

> An *aggregate* is an array or a class (Clause 12 [class]) with no user-provided constructors (15.1 [class.ctor]), **no *brace-or-equal-initializer*s for non-static data members (12.2 [class.mem]),** no private or protected non-static data members (Clause 14 [class.access]), no base classes (Clause 13 [class.derived]), and no virtual functions (13.3 [class.virtual]).

---

737. Uninitialized trailing characters in string initialization

Section: 11.6.2 [dcl.init.string] **Status:** CD2 **Submitter:** James Kanze **Date:** 26 October, 2008

[Voted into WP at October, 2009 meeting.]

The current specification of string initialization in 11.6.2 [dcl.init.string] leaves uninitialized all characters following the terminating `'\0'` of a character array with automatic storage duration. This is different from C99, in which string initialization is handled like aggregate initialization and all trailing characters are zeroed (6.7.8 paragraph 21).

(See also issue 694, in which we are considering following C99 in a somewhat similar case of zero-initializing trailing data.)

Proposed resolution (September, 2009):

Add a new paragraph following 11.6.2 [dcl.init.string] paragraph 2:

> There shall not be more initializers than there are array elements. [*Example:*
>
> ```
> char cv[4] = "asdf";     // error
> ```
>
> is ill-formed since there is no space for the implied trailing '\0'. —*end example*]
>
> **If there are fewer initializers than there are array elements, then each element not explicitly initialized shall be zero-initialized (11.6 [dcl.init]).**

---

936. Array initialization with new string literals

Section: 11.6.2 [dcl.init.string] **Status:** CD2 **Submitter:** Alisdair Meredith **Date:** 11 July, 2009

[Voted into WP at October, 2009 meeting.]

11.6.2 [dcl.init.string] paragraph 1 says,

> A `char` array (whether plain `char`, `signed char`, or `unsigned char`), `char16_t` array, `char32_t` array, or `wchar_t` array can be initialized by a *string-literal* (optionally enclosed in braces) with no prefix, with a `u` prefix, with a `U` prefix, or with an `L` prefix, respectively...

This formulation does not allow for raw and UTF-8 literals.

Proposed resolution (July, 2009):

Change 11.6.2 [dcl.init.string] paragraph 1 as follows:

> A char array (whether plain `char`, `signed char`, or `unsigned char`), `char16_t` array, `char32_t` array, or `wchar_t` array can be initialized by a ~~*string-literal* (optionally enclosed in braces) with no prefix, with a `u` prefix, with a `U` prefix, or with an `L` prefix~~ **narrow character literal,** `char16_t` **string literal,** `char32_t` **string literal, or wide string literal**, respectively~~; successive~~**, or by an appropriately-typed string literal enclosed in braces. Successive** characters of the ~~*string-literal*~~ **value of the string literal** initialize the ~~members~~ **elements** of the array. [*Example:* ...

---

589. Direct binding of class and array rvalues in reference initialization

Section: 11.6.3 [dcl.init.ref] **Status:** CD2 **Submitter:** Steve Adamczyk **Date:** 26 July 2006

[Voted into WP at October, 2009 meeting.]

The resolutions of issues 391 and 450 say that the reference is "bound to" the class or array rvalue, but it does not say that the reference "binds directly" to the initializer, as it does for the cases that fall under the first bullet in 11.6.3 [dcl.init.ref] paragraph 5. However, this phrasing is important in determining the implicit conversion sequence for an argument passed to a parameter with reference type (16.3.3.1.4 [over.ics.ref]), where paragraph 2 says,

> When a parameter of reference type is not bound directly to an argument expression, the conversion sequence is the one required to convert the argument expression to the underlying type of the reference according to 16.3.3.1 [over.best.ics]. Conceptually, this conversion sequence corresponds to copy-initializing a temporary of the underlying type with the argument expression.

The above-mentioned issue resolutions stated that no copy is to be made in such reference initializations, so the determination of the conversion sequence does not reflect the initialization semantics.

Simply using the "binds directly" terminology in the new wording may not be the right approach, however, as there are other places in the Standard that also give special treatment to directly-bound references. For example, the first bullet of 8.16 [expr.cond] paragraph 3

says,

> If E2 is an lvalue: E1 can be converted to match E2 if E1 can be implicitly converted (clause 7 [conv]) to the type "reference to T2," subject to the constraint that in the conversion the reference must bind directly (11.6.3 [dcl.init.ref]) to E1.

The effect of simply saying that a reference "binds directly" to a class rvalue can be seen in this example:

```
struct B { };
struct D: B { };
D f();
void g(bool x, const B& br) {
    x ? f() : br;   // result would be lvalue
}
```

It is not clear that treating this conditional expression as an lvalue is a desirable outcome, even if the result of `f()` were to "bind directly" to the `const B&` reference.

Proposed resolution (June, 2009):

1. Change 11.6.3 [dcl.init.ref] paragraph 5 as follows:

   A reference to type "*cv1* T1" is initialized by an expression of type "*cv2* T2" as follows:

   - If the reference is an lvalue reference and the initializer expression
     - is an lvalue (but is not a bit-field), and "*cv1* T1" is reference-compatible with "*cv2* T2," or
     - has a class type (i.e., T2 is a class type), where T1 is not reference-related to T2, and can be implicitly converted to an lvalue of type "*cv3* T3," where "*cv1* T1" is reference-compatible with "*cv3* T3" (this conversion is selected by enumerating the applicable conversion functions (16.3.1.6 [over.match.ref]) and choosing the best one through overload resolution (16.3 [over.match])),

     then the reference is bound ~~directly~~ to the initializer expression lvalue in the first case~~, and the reference is bound~~ **and** to the lvalue result of the conversion in the second case. ~~In these cases the reference is said to *bind directly* to the initializer expression.~~ [*Note:* the usual lvalue-to-rvalue (7.1 [conv.lval]), array-to-pointer (7.2 [conv.array]), and function-to-pointer (7.3 [conv.func]) standard conversions are not needed, and therefore are suppressed, when such direct bindings to lvalues are done. —*end note*]

     [*Example:* ... —*end example*]

   - Otherwise, the reference shall be an lvalue reference to a non-volatile const type (i.e., *cv1* shall be `const`), or the reference shall be an rvalue reference and the initializer expression shall be an rvalue. [*Example:* ... —*end example]*
     - If the initializer expression is an rvalue, with T2 a class type, and "*cv1* T1" is reference-compatible with "*cv2* T2," the reference is bound to the object represented by the rvalue (see 6.10 [basic.lval]) or to a sub-object within that object.

       [*Example:* ... —*end example*]

     - If the initializer expression is an rvalue, with T2 an array type, and "*cv1* T1" is reference-compatible with "*cv2* T2," the reference is bound to the object represented by the rvalue (see 6.10 [basic.lval]).
     - Otherwise, a temporary of type "*cv1* T1" is created and initialized from the initializer expression using the rules for a non-reference copy initialization (11.6 [dcl.init]). The reference is then bound to the temporary. If T1 is reference-related to T2, *cv1* must be the same cv-qualification as, or greater cv-qualification than, *cv2*; otherwise, the program is ill-formed. [*Example:* ... —*end example*]

   In all cases except the last (i.e., creating and initializing a temporary from the initializer expression), the reference is said to *bind directly* to the initializer expression.

2. Change 8.16 [expr.cond] paragraph 3 bullet 1 as follows:

   - If E2 is an lvalue: E1 can be converted to match E2 if E1 can be implicitly converted (Clause 7 [conv]) to the type "lvalue reference to T2", subject to the constraint that in the conversion the reference must bind directly (11.6.3 [dcl.init.ref]) to ~~E1~~ **an lvalue**.

---

656. Direct binding to the result of a conversion operator

Section: 11.6.3 [dcl.init.ref] **Status:** CD2 **Submitter:** Jason Merrill **Date:** 23 October 2007

[Voted into WP at October, 2009 meeting.]

Consider the following example:

```
struct A { };
struct B : public A { };
struct X {
   operator B();
};
X x;
```

```
int main() {
   const A& r = x;
   return 0;
}
```

It seems like the resolution of issue 391 doesn't actually cover this; `X` is not reference-compatible with `A`, so we go past the modified bullet (11.6.3 [dcl.init.ref] paragraph 5, bullet 2, sub-bullet 1), which reads:

> If the initializer expression is an rvalue, with `T2` a class type, and "*cv1* `T1`" is reference-compatible with "*cv2* `T2`," the reference is bound to the object represented by the rvalue (see 6.10 [basic.lval]) or to a sub-object within that object.

and hit

> Otherwise, a temporary of type "*cv1* `T1`" is created and initialized from the initializer expression using the rules for a non-reference copy initialization (11.6 [dcl.init]). The reference is then bound to the temporary.

which seems to require that we create an `A` temporary copied from the return value of `X::operator B()` rather than bind directly to the `A` subobject. I think that the resolution of issue 391 should cover this situation as well, and the EDG compiler seems to agree with me.

(See also issue 896.)

Proposed resolution (September, 2009):

1. Change 11.6.3 [dcl.init.ref] paragraph 5 as follows:
   - If the reference is an lvalue reference...
   - Otherwise, the reference shall be an lvalue reference to a non-volatile const type...
     - ~~If the initializer expression is an rvalue, with `T2` a class type, and "*cv1* `T1`" is reference-compatible with "*cv2* `T2`," the reference is bound to the object represented by the rvalue (see 6.10 [basic.lval]) or to a sub-object within that object.~~ **If `T1` and `T2` are class types and**
       - **the initializer expression is an rvalue, and "*cv1* `T1`" is reference-compatible with "*cv2* `T2`," or**
       - **`T1` is not reference-related to `T2`, and the initializer expression can be implicitly converted to an rvalue of type "*cv3* `T3`," where "*cv1* `T1`" is reference-compatible with "*cv3* `T3`" (this conversion is selected by enumerating the applicable conversion functions (16.3.1.6 [over.match.ref]) and choosing the best one through overload resolution (16.3 [over.match])),**

     then the reference is bound to the initializer expression rvalue in the first case, and to the object that is the result of the conversion in the second case (or, in either case, to the appropriate base class subobject of the object). [*Example:*

```
struct A { };
struct B : A { } b;
extern B f();
const A& rca = f();     // Bound to the A subobject of the B rvalue.
A&& rcb = f();          // Same as above
struct X {
  operator B();
} x;
const A& r = x;           // Bound to the A subobject of the result of the conversion
```

     —end example]
     - ...

   *Editorial note: issue 589 makes edits to the top-level bullet preceding this one. The wording resulting from those edits should be changed for consistency with this wording so that the text there reads, "...in the first case and to the lvalue result of the conversion in the second case **(or, in either case, to the appropriate base class subobject of the object)**."*

2. Change 16.3 [over.match] paragraph 2, last bullet as follows:
   - invocation of a conversion function for conversion to an lvalue **or class rvalue** to which a reference (11.6.3 [dcl.init.ref]) will be directly bound (16.3.1.6 [over.match.ref]).

3. Change 16.3.1.6 [over.match.ref] paragraph 1 as follows:

   > Under the conditions specified in 11.6.3 [dcl.init.ref], a reference can be bound directly to an lvalue **or class rvalue** that is the result of applying a conversion function to an initializer expression. Overload resolution is used to select the conversion function to be invoked. Assuming that "*cv1* `T`" is the underlying type of the reference being initialized, and "*cv* `S`" is the type of the initializer expression, with `S` a class type, the candidate functions are selected as follows:
   >
   > - The conversion functions of `S` and its base classes are considered, except that for copy-initialization, only the non-explicit conversion functions are considered. Those that are not hidden within `S` and yield type "lvalue reference to *cv2* `T2`" **(when 11.6.3 [dcl.init.ref] requires an lvalue result), or "*cv2* `T2`" or "rvalue reference to *cv2* `T2` (when 11.6.3 [dcl.init.ref] requires an rvalue result)**, where "*cv1* `T`" is reference-compatible (11.6.3 [dcl.init.ref]) with "*cv2* `T2`", are candidate functions.

(Note: this resolution also resolves issue 896.)

664. Direct binding of references to non-class rvalue references

Section: 11.6.3 [dcl.init.ref] **Status:** CD2 **Submitter:** Eric Niebler **Date:** 1 December 2007

[Voted into WP at March, 2010 meeting as document N3055.]

According to 11.6.3 [dcl.init.ref] paragraph 5, a reference initialized with a reference-compatible rvalue of class type binds directly to the object. A reference-compatible non-class rvalue reference, however, is first copied to a temporary and the reference binds to that temporary, not to the target of the rvalue reference. This can cause problems when the result of a forwarding function is used in such a way that the address of the result is captured. For example:

```
struct ref {
    explicit ref(int&& i): p(&i) { }
    int* p;
};

int&& forward(int&& i) {
    return i;
}

void f(int&& i) {
    ref r(forward(i));
    // Here r.p is a dangling pointer, pointing to a defunct int temporary
}
```

A formulation is needed so that rvalue references are treated like class and array rvalues.

Notes from the February, 2008 meeting:

You can't just treat scalar rvalues like class and array rvalues, because they might not have an associated object. However, if you have an rvalue reference, you know that there is an object, so probably the best way to address this issue is to specify somehow that binding a reference to an rvalue reference does not introduce a new temporary.

(See also issues 690 and 846.)

Proposed resolution (February, 2010):

See paper N3030.

---

896. Rvalue references and rvalue-reference conversion functions

Section: 11.6.3 [dcl.init.ref] **Status:** CD2 **Submitter:** Steve Adamczyk **Date:** 9 May, 2009

[Voted into WP at October, 2009 meeting.]

Consider the following example:

```
struct A { } a;
struct B {
  operator A&&() {
    return static_cast<A&&>(a);
  }
};
A&& r = B();
```

One would expect that `r` would be bound to the object returned by `B::operator A&&()`, i.e., `a`. However, the logic in 11.6.3 [dcl.init.ref] paragraph 5 requires that the result of the conversion function be copied to a temporary and `r` bound to the temporary.

Probably the way to address this is to add another top-level bullet between the first and second that would essentially mimic the first bullet except dealing with rvalue references: direct binding to reference-compatible rvalues or to the reference-compatible result of a conversion function. (Note that this should only apply to class rvalues; the creation of a temporary for non-class rvalues is necessary to have an object for the reference to bind to.)

(See also issue 656.)

Proposed resolution (September, 2009):

This issue is resolved by the resolution of issue 656.

---

703. Narrowing for literals that cannot be exactly represented

Section: 11.6.4 [dcl.init.list] **Status:** CD2 **Submitter:** Jason Merrill **Date:** 2 July, 2008

[Voted into WP at October, 2009 meeting.]

Both of the following initializations are ill-formed because of narrowing, although they were previously well-formed:

```
struct A { int i; } a = { 1.0 };
struct B { float f; } b = { 1.1 };
```

The first one doesn't seem like a big problem, as there probably isn't much code that has this kind of aggregate initialization. The second might be of more concern, because `1.1` is not representable in either `float` or `double`. Is the resulting loss of precision a kind of narrowing that we want to diagnose?

Notes from the September, 2008 meeting:

The CWG agreed that the second initialization should not be a narrowing error; furthermore, this exemption should apply not only to literals but to any floating-point constant expression. Instead of the current formulation, requiring exact bidirectional convertibility, the Standard should only require that the initializer value be within the representable range of the target type.

Proposed resolution (July, 2009):

Change 11.6.4 [dcl.init.list] paragraph 6 as follows:

> A *narrowing conversion* is an implicit conversion
>
> - from a floating-point type to an integer type, or
> - from `long double` to `double` or `float`, or from `double` to `float`, except where the source is a constant expression and the actual value after conversion ~~will fit into the target type and will produce the original value when converted back to the original type~~ **is within the range of values that can be represented (even if it cannot be represented exactly)**, or
> - ...

---

865. Initializing a `std::initializer_list`

Section: 11.6.4 [dcl.init.list] **Status:** CD2 **Submitter:** James Widman **Date:** 8 April, 2009

[Voted into WP at October, 2009 meeting.]

There are several problems with the wording of 11.6.4 [dcl.init.list] paragraph 4:

> When an initializer list is implicitly converted to a `std::initializer_list<E>`, the object passed is constructed as if the implementation allocated an array of *N* elements of type `E`, where *N* is the number of elements in the initializer list. Each element of that array is initialized with the corresponding element of the initializer list converted to `E`, and the `std::initializer_list<E>` object is constructed to refer to that array. If a narrowing conversion is required to convert the element to `E`, the program is ill-formed.

First, an initializer list is not an expression, so it is not appropriate to refer to "implicitly convert[ing]" it, as is done in the first sentence.

Also, the conversion of the elements of the initializer list to the elements of the array is not specified to be either copy-initialization or direct-initialization. If this is intended to be viewed as an aggregate initialization, it would be copy-initialization, but that needs to be specified more clearly.

Finally, the initializer list can have nested initializer lists, so the references to converting the element also need to be cleaned up.

Proposed resolution (July, 2009):

Change 11.6.4 [dcl.init.list] paragraph 4 as follows:

> ~~When an initializer list is implicitly converted to a~~ **An object of type** `std::initializer_list<E>` **is constructed from an initializer list**~~, the object passed is constructed~~ as if the implementation allocated an array of *N* elements of type `E`, where *N* is the number of elements in the initializer list. Each element of that array is **copy-**initialized with the corresponding element of the initializer list~~ converted to `E`~~, and the `std::initializer_list<E>` object is constructed to refer to that array. If a narrowing conversion is required to ~~convert the element to `E`~~ **initialize any of the elements**, the program is ill-formed. [*Example:*...

---

934. List-initialization of references

Section: 11.6.4 [dcl.init.list] **Status:** CD2 **Submitter:** Mike Miller **Date:** 8 July, 2009

[Voted into WP at October, 2009 meeting.]

According to 11.6.4 [dcl.init.list] paragraph 3,

> Otherwise, if `T` is a reference type, an rvalue temporary of the type referenced by `T` is list-initialized, and the reference is bound to that temporary.

This means, for an example like

```
int i;
const int& r1{ i };
int&& r2{ i };
```

`r1` is bound to a temporary containing the value of `i`, not to `i` itself, which seems surprising. Also, there's no prohibition here against binding the rvalue reference to an lvalue, as there is in 11.6.3 [dcl.init.ref] paragraph 5 bullet 2, so the initialization of `r2` is well-formed, even though the corresponding non-list initialization `int&& r3(i)` is ill-formed.

There's also a question as to whether this bullet even applies to these examples. According to the decision tree in 11.6 [dcl.init] paragraph 16, initialization of a reference is dispatched to 11.6.3 [dcl.init.ref] in the first bullet, so these cases never make it to the third bullet sending the remaining braced-init-list cases to 11.6.4 [dcl.init.list]. If that's the correct interpretation, there's a problem with 11.6.3 [dcl.init.ref], since it doesn't deal with the *braced-init-list* cases, and the bullet in 11.6.4 [dcl.init.list] paragraph 3 dealing with references is dead code that's never used.

Proposed resolution (July, 2009):

1. Move the third bullet of the list in 11.6 [dcl.init] paragraph 16 to the top of the list:
   - **If the initializer is a *braced-init-list*, the object is list-initialized (11.6.4 [dcl.init.list]).**
   - If the destination type is a reference type, see 11.6.3 [dcl.init.ref].
   - ...
2. Change 11.6.4 [dcl.init.list] paragraph 3, bullets 4 and 5, as follows:
   - Otherwise, if `T` is a reference **to class** type, **or if `T` is any reference type and the initializer list has no elements,** an rvalue temporary of the type referenced by `T` is list-initialized, and the reference is bound to that temporary. [*Note:*...
   - Otherwise ~~(i.e., if `T` is not an aggregate, class type, or reference)~~, if the initializer list has a single element...

---

989. Misplaced list-initialization example

Section: 11.6.4 [dcl.init.list] **Status:** CD2 **Submitter:** Daniel Krügler **Date:** 20 October, 2009

[Voted into WP at March, 2010 meeting.]

The final set of declarations in the example following 11.6.4 [dcl.init.list] paragraph 3 bullet 3 is:

```
struct S2 {
  int m1;
  double m2,m3;
};
S2 s21 = { 1, 2, 3.0 };    // OK
S2 s22 { 1.0, 2, 3 };      // error: narrowing
S2 s23 {};                 // OK: default to 0,0,0
```

However, `S2` is an aggregate. Aggregates are handled in bullet 1, while bullet 3 deals with classes with constructors. This part of the example should be moved to the first bullet.

Proposed resolution (October, 2009):

Move the `S2` example from bullet 3 to bullet 1 in 11.6.4 [dcl.init.list] paragraph 3:

- If `T` is an aggregate, aggregate initialization is performed (11.6.1 [dcl.init.aggr]).

  [*Example:*

  ```
  double ad[] = { 1, 2.0 };    // OK
  int ai[] = { 1, 2.0 };       // error: narrowing

  struct S2 {
    int m1;
    double m2,m3;
  };
  S2 s21 = { 1, 2, 3.0 };      // OK
  S2 s22 { 1.0, 2, 3 };        // error: narrowing
  S2 s23 {};                   // OK: default to 0,0,0
  ```

  —*end example*]

- Otherwise, if `T` is a specialization...
- Otherwise, if T is a class type...

  [*Example:*

  ```
  ...
  S s3 { };                    // OK: invoke #2

  ~~struct S2 {~~
  ~~  int m1;~~
  ~~  double m2,m3;~~
  ~~};~~
  ~~S2 s21 = { 1, 2, 3.0 };      //OK~~
  ~~S2 s22 { 1.0, 2, 3 };        //error: narrowing~~
  ~~S2 s23 {};                   //OK: default to 0,0,0~~
  ```

—end example]

- ...

---

990. Value initialization with multiple initializer-list constructors

Section: 11.6.4 [dcl.init.list] **Status:** CD2 **Submitter:** Daniel Krügler **Date:** 20 October, 2009

[Voted into WP at March, 2010 meeting as part of document N3079.]

It should always be possible to use the new brace syntax to value-initialize an object. However, the current rules make the following example ill-formed because of ambiguity:

```
struct S {
  S();
  S(std::initializer_list<int>);
  S(std::initializer_list<double>);
};
S s{};     // Ambiguous initializer-list constructor reference,
           // not value initialization.
```

Proposed resolution (February, 2010):

Change 11.6.4 [dcl.init.list] paragraph 3 as follows:

List-initialization of an object or reference of type `T` is defined as follows:

- **If the initializer list has no elements and `T` is a class type with a default constructor, the object is value-initialized.**
- **Otherwise, if the initializer list has no elements and `T` is an aggregate, the initializer list is used to initialize each of the members of `T`. [*Example:***

```
struct A {
  A(std::initializer_list<int>);  // #1
};
struct B {
  A a;
};
B b { };     // OK, uses #1
B b { 1 };   // error
```

***—end example*]**

- ~~If~~ **Otherwise, if** `T` is an aggregate...
- ...

[*Example:*

```
struct S {
  S(std::initializer_list<double>); // #1
  S(std::initializer_list<int>);    // #2
  S();                              // #3
  // ...
};
S s1 = { 1.0, 2.0, 3.0 };           // invoke #1
S s2 = { 1, 2, 3 };                 // invoke #2
S s3 = { };                         // invoke #3 (for value-initialization; see above)
```

—end example]

---

905. Explicit defaulted copy constructors and trivial copyability

Section: 12 [class] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 27 May, 2009

[Voted into WP at March, 2010 meeting.]

It is presumably possible to declare a defaulted copy constructor to be `explicit`. Should that render a class not trivially copyable, even though the copy constructor is trivial? That is, does being "trivally copyable" mean that copy initialization, and not just direct initialization, is possible?

A related question is whether the specification of triviality should require that the copy constructor and copy assignment operator must be public. (With the advent of "`=default`" it is possible to make them non-public, which was not the case when these definitions were crafted.)

Proposed resolution (October, 2009):

This issues is resolved by the resolution of issue 906.

---

645. Are bit-field and non-bit-field members layout compatible?

Section: 12.2 [class.mem] **Status:** CD2 **Submitter:** Alan Stokes **Date:** 9 Aug 2007

[Voted into the WP at the March, 2009 meeting.]

The current wording defining a "common initial sequence" in 12.2 [class.mem] paragraph 17 does not address the case in which one member is a bit-field and the corresponding member is not:

> Two standard-layout structs share a common initial sequence if corresponding members have layout-compatible types (and, for bit-fields, the same widths) for a sequence of one or more initial members.

Presumably the intent was something like, "(and, if one of the pair is a bit-field, the other is also a bit-field of the same width)."

Proposed Resolution (September, 2008):

Change 12.2 [class.mem] paragraph 18 as follows:

> ... Two standard-layout structs share a common initial sequence if corresponding members have layout-compatible types ~~(and, for bit-fields, the same widths)~~ **and either neither member is a bit-field or both are bit-fields with the same widths** for a sequence of one or more initial members.

---

874. Class-scope definitions of enumeration types

Section: 12.2 [class.mem] **Status:** CD2 **Submitter:** Daniel Krügler **Date:** 16 April, 2009

[Voted into WP at October, 2009 meeting.]

According to 12.2 [class.mem] paragraph 1,

> The enumerators of an enumeration (10.2 [dcl.enum]) defined in the class are members of the class... A member shall not be declared twice in the *member-specification*, except that a nested class or member class template can be declared and then later defined.

The enumerators of a scoped enumeration are not members of the containing class; the wording should be revised to apply only to unscoped enumerations.

The second part of the cited wording from 12.2 [class.mem] prohibits constructs like:

```
class C {
  public:
    enum E: int;
  private:
    enum E: int { e0 };
};
```

which might be useful in making the enumeration type, but not its enumerators, accessible.

Notes from the July, 2009 meeting:

According to 14.1 [class.access.spec] paragraph 4, the access must be the same for all declarations of a class member. The suggested usage given above violates that requirement: the second declaration of `E` declares the enumeration itself, not just the enumerators, to be private. The CWG did not feel that the utility of the suggested feature warranted the complexity of an exception to the general rule.

Proposed resolution (July, 2009):

1. Change 12.2 [class.mem] paragraph 1 as follows:

   > ...The enumerators of an **unscoped** enumeration (10.2 [dcl.enum]) defined in the class are members of the class... A member shall not be declared twice in the *member-specification*, except that a nested class or member class template can be declared and then later defined**, and except that an enumeration can be first introduced with an *opaque-enum-declaration* and then later be redeclared with an *enum-specifier***.

2. Change the example in 14.1 [class.access.spec] paragraph 4 as follows:

   When a member is redeclared within its class definition, the access specified at its redeclaration shall be the same as at its initial declaration. [*Example:*

```
struct S {
  class A;
   enum E : int;
private:
  class A { };          // error: cannot change access
   enum E : int { e0 };  // error: cannot change access
};
```

—end example]

---

714. Static const data members and *braced-init-list*s

Section: 12.2.3.2 [class.static.data] **Status:** CD2 **Submitter:** Steve Adamczyk **Date:** 15 September, 2008

[Voted into WP at July, 2009 meeting.]

The recent changes in the handling of initialization have not touched the requirement that the in-class initializer for a const static data member must be of the form `=` *assignment-expression* and not a *braced-init-list*. It would be more consistent and general to allow the braced form as well.

Proposed resolution (March, 2009):

1. Change 8.20 [expr.const] paragraph 3 as follows:

   ...as enumerator initializers (10.2 [dcl.enum]), ~~as static member initializers (12.2.3.2 [class.static.data]),~~ and as integral or enumeration non-type template arguments (17.5 [temp.type]).

2. Change 12.2.3.2 [class.static.data] paragraph 3 as follows:

   If a `static` data member is of `const` effective literal type, its declaration in the class definition can specify a *brace-or-equal-initializer* ~~with an~~ **in which every** *initializer-clause* that is an ***assignment-expression*** **is a** ~~integral~~ constant expression. A `static` data member of effective literal type can be declared in the class definition with the `constexpr` specifier; if so, its declaration shall specify a *brace-or-equal-initializer* ~~with an~~ **in which every** *initializer-clause* that is an ***assignment-expression*** **is a** ~~integral~~ constant expression. **[*Note:* In both these cases, the member may appear in ~~integral~~ constant expressions. *—end note*]** The member shall still be defined in a namespace scope if it is used in the program and the namespace scope definition shall not contain an initializer.

[Drafting note: this change also corrects an editorial error resulting from overlapping changes that inadvertently retained the original restriction that only members of integral type could be initialized inside the class definition.]

---

716. Specifications that should apply only to non-static union data members

Section: 12.3 [class.union] **Status:** CD2 **Submitter:** Mike Miller **Date:** 17 September, 2008

[Voted into WP at July, 2009 meeting.]

Unions are no longer forbidden to have static data members; however, much of the wording of 12.3 [class.union] (and possibly other places in the Standard) is still written with that assumption and refers only to "data members" when clearly non-static data members are in view. From paragraph 1, for example:

> In a union, at most one of the **data members** can be active at any time... The size of a union is sufficient to contain the largest of its **data members**...

Proposed resolution (March, 2009):

1. Change the footnote in 6.9.3 [basic.type.qualifier] paragraph 1 as follows:

   The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and **non-static data** members of unions.

2. Change 6.10 [basic.lval] paragraph 15 bullet 6 as follows:

   - an aggregate or union type that includes one of the aforementioned types among its **elements or non-static data** members (including, recursively, ~~a~~ **an element or non-static data** member of a subaggregate or contained union),

3. Change 8.9 [expr.rel] paragraph 2 bullet 5 as follows:

   - If two pointers point to **non-static** data members of the same union object, they compare equal (after conversion to `void*`, if necessary)...

4. Change 10.6.2 [dcl.align] paragraph 8 as follows:

   [*Note:* the alignment of a union type can be strengthened by applying the alignment attribute to any **non-static data** member of the union. *—end note*]

5. Change 11.6.1 [dcl.init.aggr] paragraph 15 as follows:

   When a union is initialized with a brace-enclosed initializer, the braces shall only contain an *initializer-clause* for the first **non-static data** member of the union...

6. Change 12.3 [class.union] paragraph 1 as follows:

> In a union, at most one of the **non-static** data members can be active at any time, that is, the value of at most one of the **non-static** data members can be stored in a union at any time. [*Note:* one special guarantee is made in order to simplify the use of unions: If a standard-layout union contains several standard-layout structs that share a common initial sequence (12.2 [class.mem]), and if an object of this standard-layout union type contains one of the standard-layout structs, it is permitted to inspect the common initial sequence of any of standard-layout struct members; see 12.2 [class.mem]. —*end note*] The size of a union is sufficient to contain the largest of its **non-static** data members. Each **non-static** data member is allocated as if it were the sole member of a struct. A union can have...

---

608. Determining the final overrider of a virtual function

Section: 13.3 [class.virtual] **Status:** CD2 **Submitter:** Mike Miller **Date:** 7 December 2006

[Voted into WP at October, 2009 meeting.]

According to 13.3 [class.virtual] paragraph 2:

> Then in any well-formed class, for each virtual function declared in that class or any of its direct or indirect base classes there is a unique *final overrider* that overrides that function and every other overrider of that function. The rules for member lookup (13.2 [class.member.lookup]) are used to determine the final overrider for a virtual function in the scope of a derived class but ignoring names introduced by *using-declaration*s.

I think that description is wrong on at least a couple of counts. First, consider the following example:

```
struct A { virtual void f(); };
struct B: A { };
struct C: A { void f(); };
struct D: B, C { };
```

What is the "unique final overrider" of `A::f()` in `D`? According to 13.3 [class.virtual] paragraph 2, we determine that by looking up `f` in `D` using the lookup rules in 13.2 [class.member.lookup]. However, that lookup determines that `f` in `D` is ambiguous, so there is no "unique final overrider" of `A::f()` in `D`. Consequently, because "any well-formed class" must have such an overrider, `D` must be ill-formed.

Of course, we all know that `D` is *not* ill-formed. In fact, 13.3 [class.virtual] paragraph 10 contains an example that illustrates exactly this point:

```
struct A {
    virtual void f();
};
struct B1 : A {      // note non-virtual derivation
    void f();
};
struct B2 : A {
    void f();
};
struct D : B1, B2 { // D has two separate A subobjects
};
```

> In class `D` above there are two occurrences of class `A` and hence two occurrences of the virtual member function `A::f`. The final overrider of `B1::A::f` is `B1::f` and the final overrider of `B2::A::f` is `B2::f`.

It appears that the requirement for a "unique final overrider" in 13.3 [class.virtual] paragraph 2 needs to say something about sub-objects. Whatever that "something" is, you can't just say "look up the name in the derived class using 13.2 [class.member.lookup]."

There's another problem with using the 13.2 [class.member.lookup] lookup to specify the final overrider: name lookup just looks up the name, while the overriding relationship is based not only on the name but on a matching parameter-type-list and cv-qualification. To illustrate this point:

```
struct X {
    virtual void f();
};
struct Y: X {
    void f(int);
};
struct Z: Y { };
```

What is the "unique final overrider" of `X::f()` in `A`? Again, 13.3 [class.virtual] paragraph 2 says you're supposed to look up `f` in `Z` to find it; however, what you find is `Y::f(int)`, not `X::f()`, and that's clearly wrong.

Proposed Resolution (December, 2006):

Change 13.3 [class.virtual] paragraph 2 as follows:

> ~~Then in any well-formed class, for each virtual function declared in that class or any of its direct or indirect base classes there is a unique *final overrider* that overrides that function and every other overrider of that function. The rules for member lookup (13.2 [class.member.lookup]) are used to determine the final overrider for a virtual function in the scope of a derived class but ignoring names introduced by *using-declaration*s.~~ **A virtual member function `vf` of a class `C` is a *final overrider* unless the most derived class (4.5 [intro.object]) of which `C` is a base class (if any) declares or inherits another member function that overrides `vf`. In a derived class, if a virtual member function of a base class subobject has more than one final overrider, the program is ill-formed.**

Proposed resolution (July, 2009):

Change 13.3 [class.virtual] paragraph 2 as follows:

> ~~...Then in any well-formed class, for each virtual function declared in that class or any of its direct or indirect base classes there is a unique *final overrider* that overrides that function and every other overrider of that function. The rules for member lookup (13.2 [class.member.lookup]) are used to determine the final overrider for a virtual function in the scope of a derived class but ignoring names introduced by *using-declarations*.~~ **A virtual member function `C::vf` of a class object `S` is a *final overrider* unless the most derived class (4.5 [intro.object]) of which `S` is a base class subobject (if any) declares or inherits another member function that overrides `vf`. In a derived class, if a virtual member function of a base class subobject has more than one final overrider, the program is ill-formed.** [*Example:* ... *—end example*] **[*Example:***
>
> ```
> struct A { virtual void f(); };
> struct B: A { };
> struct C: A { void f(); };
> struct D: B, C { };     // OK; A::f and C::f are the final overriders
>                         // for the B and C subobjects, respectively
> ```
>
> ***—end example*]**

---

939. Explicitly checking virtual function overriding

Section: 13.3 [class.virtual] **Status:** CD2 **Submitter:** FI/US **Date:** 14 July, 2009

N2800 comment FI 1
N2800 comment US 41

[Voted into WP at July, 2009 meeting as N2928.]

There should be a way to detect errors in overriding a virtual function.

Proposed resolution (July, 2009):

This issue is resolved by paper PL22.16/09-0118 = WG21 N2928.

---

960. Covariant functions and lvalue/rvalue references

Section: 13.3 [class.virtual] **Status:** CD2 **Submitter:** James Widman **Date:** 1 September, 2009

[Voted into WP at March, 2010 meeting.]

13.3 [class.virtual] paragraph 5 requires that covariant return types be either both pointers or both references, but it does not specify that references must be both lvalue references or both rvalue references. Presumably this is an oversight.

Proposed resolution (February, 2010):

Change 13.3 [class.virtual] paragraph 5 bullet 1 as follows:

> ...If a function `D::f` overrides a function `B::f`, the return types of the functions are covariant if they satisfy the following criteria:
>
> - both are pointers to classes**, both are lvalue references to classes,** or **both are rvalue** references to classes[106]
> - ...

---

922. Implicit default constructor definitions and `const` variant members

Section: 15.1 [class.ctor] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 19 June, 2009

[Voted into WP at March, 2010 meeting.]

According to 15.1 [class.ctor] paragraph 5,

> An implicitly-declared default constructor for class `X` is defined as deleted if: ... any non-static data member of const-qualified type (or array thereof) does not have a user-provided default constructor, or...

It is not clear if this adequately covers the case in which some variant members are const-qualified but others are not. The intent of the restriction is to prevent creation of an object with uninitialized members that would require a `const_cast` to set their value later, but const-qualified members of an anonymous union in which other members are not const do not seem to present that problem.

Proposed resolution (October, 2009):

Change 15.1 [class.ctor] paragraph 5 bullet 3 of the second list and add a fourth bullet as follows:

- ...
- any **non-variant** non-static data member of const-qualified type (or array thereof) does not have a user-provided default constructor, ~~or~~
- **all variant members are of const-qualified type (or array thereof), or**
- ...

Proposed resolution (November, 2009):

Change 15.1 [class.ctor] paragraph 5 bullet 3 of the second list and add two bullets as follows:

- ...
- any **non-variant** non-static data member of const-qualified type (or array thereof) does not have a user-provided default constructor, ~~or~~
- `X` **is a union and all its variant members are of const-qualified type (or array thereof),**
- `X` **is a non-union class and all members of any anonymous union member are of const-qualified type (or array thereof), or**
- ...

---

927. Implicitly-deleted default constructors and member initializers

Section: 15.1 [class.ctor] **Status:** CD2 **Submitter:** Alisdair Meredith **Date:** 1 July, 2009

[Voted into WP at March, 2010 meeting.]

(From message 14555.)

The reasons for which an implicitly-declared default constructor is defined as deleted, given in 15.1 [class.ctor] paragraph 4, all deal with cases in which a member cannot be default-initialized. Presumably a *brace-or-equal-initializer* for such a member would eliminate the need to define the constructor as deleted, but this case is not addressed by the current wording.

Proposed resolution (October, 2009):

Change 15.1 [class.ctor] paragraph 5, the second list, as follows:

> An implicitly-declared default constructor for class `X` is defined as deleted if:
>
> - `X` is a union-like class that has a variant member with a non-trivial default constructor,
> - any non-static data member **with no *brace-or-equal-initializer*** is of reference type,
> - any non-static data member of const-qualified type (or array thereof) **with no *brace-or-equal-initializer*** does not have a user-provided default constructor, or
> - any **direct or virtual base class, or** non-static data member **with no *brace-or-qual-initializer,*** ~~or direct or virtual base class~~ has class type `M` (or array thereof) and **either** `M` has no default ~~constructor, or if~~ **constructor or** overload resolution (16.3 [over.match]) as applied to `M`'s default constructor~~,~~ results in an ambiguity or **in** a function that is deleted or inaccessible from the implicitly-declared default constructor.

---

650. Order of destruction for temporaries bound to the returned value of a function

Section: 15.2 [class.temporary] **Status:** CD2 **Submitter:** Mike Miller **Date:** 14 Aug 2007

[Voted into the WP at the March, 2009 meeting.]

In describing the order of destruction of temporaries, 15.2 [class.temporary] paragraphs 4-5 say,

> There are two contexts in which temporaries are destroyed at a different point than the end of the full-expression...
>
> The second context is when a reference is bound to a temporary... A temporary bound to the returned value in a function return statement (9.6.3 [stmt.return]) persists until the function exits.

The following example illustrates the issues here:

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

S& f() {
    S s;            // #1
    return
```

```
        (S(),       // #2
         S());      // #3
}
```

If the return type of `f()` were simply `S` instead of `S&`, the two temporaries would be destroyed at the end of the full-expression in the `return` statement in reverse order of their construction, followed by the destruction of the variable `s` at block-exit, i.e., the order of destruction of the `S` objects would be #3, #2, #1.

Because the temporary #3 is bound to the returned value, however, its lifetime is extended beyond the end of the full-expression, so that `S` object #2 is destroyed before #3.

There are two problems here. First, it is not clear what "until the function exits" means. Does it mean that the temporary is destroyed as part of the normal block-exit destructions, as described in 9.6 [stmt.jump] paragraph 2:

> On exit from a scope (however accomplished), destructors (15.4 [class.dtor]) are called for all constructed objects with automatic storage duration (6.7.3 [basic.stc.auto]) (named objects or temporaries) that are declared in that scope, in the reverse order of their declaration.

Or is the point of destruction for #3 *after* the destruction of the "constructed objects... that are *declared* [emphasis mine] in that scope" (because temporary #3 was not "declared")? I.e., should #3 be destroyed before or after #1?

The other problem is that, according to the recollection of one of the participants responsible for this wording, the intent was not to extend the lifetime of #3 but simply to emphasize that its lifetime ended before the function returned, i.e., that the result of `f()` could not be used without causing undefined behavior. This is also consistent with the treatment of this example by many implementations; MSVC++, g++, and EDG all destroy #3 before #2.

Suggested resolution:

Change 15.2 [class.temporary] paragraph 5 as indicated:

> ~~A~~ **The lifetime of a** temporary bound to the returned value in a function return statement (9.6.3 [stmt.return]) ~~persists until the function exits~~ **is not extended; it is destroyed at the end of the full-expression in the return statement**.

Proposed resolution (June, 2008):

Change 15.2 [class.temporary] paragraph 5 as follows (converting the running text into a bulleted list and making the indicated edits to the wording):

> ... The temporary to which the reference is bound or the temporary that is the complete object of a subobject to which the reference is bound persists for the lifetime of the reference except**:** ~~as specified below.~~
>
> - A temporary bound to a reference member in a constructor's ctor-initializer (15.6.2 [class.base.init]) persists until the constructor exits.
> - A temporary bound to a reference parameter in a function call (8.2.2 [expr.call]) persists until the completion of the full expression containing the call.
> - ~~A~~ **The lifetime of a** temporary bound to the returned value in a function return statement (9.6.3 [stmt.return]) ~~persists until the function exits~~ **is not extended; the temporary is destroyed at the end of the full-expression in the return statement.**
>
> The destruction of a temporary whose lifetime is not extended...

---

542. Value initialization of arrays of POD-structs

Section: 15.6 [class.init] **Status:** CD2 **Submitter:** Alisdair Meredith **Date:** 27 October 2005

[Voted into the WP at the March, 2009 meeting.]

15.6 [class.init] paragraph 2 says,

> When an array of class objects is initialized (either explicitly or implicitly), the constructor shall be called for each element of the array, following the subscript order;

That implies that, given

```
struct POD {
  int x;
};

POD data[10] = {};
```

this should call the implicitly declared default ctor 10 times, leaving 10 uninitialized ints, rather than value initialize each member of data, resulting in 10 initialized ints (which is required by 11.6.1 [dcl.init.aggr] paragraph 7).

I suggest rephrasing along the lines:

> When an array is initialized (either explicitly or implicitly), each element of the array shall be initialized in turn, following the subscript order;

This would allow for PODs and other classes with a dual nature under value/default initialization, and cover copy initialization for arrays too.

Proposed resolution (October, 2006):

Change 15.6 [class.init] paragraph 3 as follows:

> When an array of class objects is initialized (either explicitly or implicitly) **and the elements are initialized by constructor**, the constructor shall be called for each element of the array, following the subscript order; see 11.3.4 [dcl.array].

---

257. Abstract base constructors and virtual base initialization

Section: 15.6.2 [class.base.init] **Status:** CD2 **Submitter:** Mike Miller **Date:** 1 Nov 2000

[Voted into WP at October, 2009 meeting.]

Must a constructor for an abstract base class provide a *mem-initializer* for each virtual base class from which it is directly or indirectly derived? Since the initialization of virtual base classes is performed by the most-derived class, and since an abstract base class can never be the most-derived class, there would seem to be no reason to require constructors for abstract base classes to initialize virtual base classes.

It is not clear from the Standard whether there actually is such a requirement or not. The relevant text is found in 15.6.2 [class.base.init] paragraph 6:

> All sub-objects representing virtual base classes are initialized by the constructor of the most derived class (4.5 [intro.object]). If the constructor of the most derived class does not specify a *mem-initializer* for a virtual base class `V`, then `V`'s default constructor is called to initialize the virtual base class subobject. If `V` does not have an accessible default constructor, the initialization is ill-formed. A *mem-initializer* naming a virtual base class shall be ignored during execution of the constructor of any class that is not the most derived class.

This paragraph requires only that the most-derived class's constructor have a *mem-initializer* for virtual base classes. Should the silence be construed as permission for constructors of classes that are not the most-derived to omit such *mem-initializer*s?

Christopher Lester, on comp.std.c++, March 19, 2004: If any of you reading this posting happen to be members of the above working group, I would like to encourage you to review the suggestion contained therein, as it seems to me that the final tenor of the submission is both (a) correct (the silence of the standard DOES mandate the omission) and (b) describes what most users would intuitively expect and desire from the C++ language as well.

The suggestion is to make it clearer that constructors for abstract base classes should not be required to provide initialisers for any virtual base classes they contain (as only the most-derived class has the job of initialising virtual base classes, and an abstract base class cannot possibly be a most-derived class).

For example:

```
struct A {
  A(const int i, const int j) {};
};

struct B1 : virtual public A {
  virtual void moo()=0;
  B1() {};   // (1) Look! not "B1() : A(5,6) {};"
};

struct B2 : virtual public A {
  virtual void cow()=0;
  B2() {};   // (2) Look! not "B2() : A(7,8) {};"
};

struct C : public B1, public B2 {
  C() : A(2,3) {};
  void moo() {};
  void cow() {};
};

int main() {
  C c;
  return 0;
};
```

I believe that, by not expressly forbidding it, the standard does (and should!) allow the above code. However, as the standard doesn't expressly allow it either (have I missed something?) there appears to be room for misunderstanding. For example, g++ version 3.2.3 (and maybe other versions as well) rejects the above code with messages like:

```
In constructor `B1::B1()':
no matching function for call to `A::A()'
candidates are: A::A(const A&)
                A::A(int, int)
```

Fair enough, the standard is perhaps not clear enough. But it seems to be a shame that although this issue was first raised in 2000, we are still living with it today.

Note that we can work-around, and persuade g++ to compile the above by either (a) providing a default constructor A() for A, or (b) supplying default values for i and j in A(i,j), or (c) replace the construtors B1() and B2() with the forms shown in the two comments in the

above example.

All three of these workarounds may at times be appropriate, but equally there are other times when all of these workarounds are particularly bad. (a) and (b) may be very bad if you are trying to enforce string contracts among objects, while (c) is just barmy (I mean why did I have to invent random numbers like 5, 6, 7 and 8 just to get the code to compile?).

So to to round up, then, my plea to the working group is: "at the very least, please make the standard clearer on this issue, but preferrably make the decision to expressly allow code that looks something like the above"

Proposed resolution (July, 2009):

1. Add the indicated text (moved from paragraph 11) to the end of 15.6.2 [class.base.init] paragraph 7:

   ...The initialization of each base and member constitutes a full-expression. Any expression in a *mem-initializer* is evaluated as part of the full-expression that performs the initialization. **A *mem-initializer* where the *mem-initializer-id* names a virtual base class is ignored during execution of a constructor of any class that is not the most derived class.**

2. Change 15.6.2 [class.base.init] paragraph 8 as follows:

   If a given non-static data member or base class is not named by a *mem-initializer-id* (including the case where there is no *mem-initializer-list* because the constructor has no *ctor-initializer*) **and the entity is not a virtual base class of an abstract class (13.4 [class.abstract])**, then

   - if the entity is a non-static data member that has a *brace-or-equal-initializer*, the entity is initialized as specified in 11.6 [dcl.init];
   - otherwise, if the entity is a variant member (12.3 [class.union]), no initialization is performed;
   - otherwise, the entity is default-initialized (11.6 [dcl.init]).

   [*Note:* An abstract class (13.4 [class.abstract]) is never a most derived class, thus its constructors never initialize virtual base classes, therefore the corresponding *mem-initializer*s may be omitted. —*end note*] After the call to a constructor for class `X` has completed...

3. Change 15.6.2 [class.base.init] paragraph 10 as follows:

   Initialization ~~shall proceed~~ **proceeds** in the following order:

   - First, and only for the constructor of the most derived class ~~as described below~~ **(4.5 [intro.object])**, virtual base classes ~~shall be~~ **are** initialized in the order they appear on a depth-first left-to-right traversal of the directed acyclic graph of base classes, where "left-to-right" is the order of appearance of the base class names in the derived class *base-specifier-list*.
   - Then, direct base classes ~~shall be~~ **are** initialized in declaration order as they appear in the *base-specifier-list* (regardless of the order of the *mem-initializer*s).
   - Then, non-static data members ~~shall be~~ **are** initialized in the order they were declared in the class definition (again regardless of the order of the *mem-initializer*s).
   - Finally, the *compound-statement* of the constructor body is executed.

   [*Note:* the declaration order is mandated to ensure that base and member subobjects are destroyed in the reverse order of initialization. —*end note*]

4. Remove all normative text in 15.6.2 [class.base.init] paragraph 11, keeping the example:

   ~~All subobjects representing virtual base classes are initialized by the constructor of the most derived class (4.5 [intro.object]). If the constructor of the most derived class does not specify a *mem-initializer* for a virtual base class `V`, then `V`'s default constructor is called to initialize the virtual base class subobject. If `V` does not have an accessible default constructor, the initialization is ill-formed. A *mem-initializer* naming a virtual base class shall be ignored during execution of the constructor of any class that is not the most derived class.~~ [*Example:*...

---

888. Union member initializers

Section: 15.6.2 [class.base.init] **Status:** CD2 **Submitter:** Alisdair Meredith **Date:** 6 May, 2009

[Voted into WP at October, 2009 meeting.]

15.6.2 [class.base.init] paragraph 5 forbids initializing multiple members of a union via *mem-initializer*s:

> If a *ctor-initializer* specifies more than one *mem-initializer* for the same member, for the same base class or for multiple members of the same union (including members of anonymous unions), the *ctor-initializer* is ill-formed.

However, there is no corresponding restriction against specifying *brace-or-equal-initializer*s for multiple union members, nor for a non-overlapping pair of *brace-or-equal-initializer* and *mem-initializer*. This is presumably an oversight.

Proposed resolution (July, 2009):

1. Change 12.3 [class.union] paragraph 1 as follows:

   ...If a union contains a non-static data member of reference type the program is ill-formed. **At most one non-static data member of a union shall have a *brace-or-equal-initializer*.** [*Note:*...

2. Change 15.6.2 [class.base.init] paragraph 5 as follows:

   ...If a *ctor-initializer* specifies more than one *mem-initializer* for the same member~~,~~ **or** for the same base class ~~or for multiple members of the same union (including members of anonymous unions)~~, the *ctor-initializer* is ill-formed.

3. Change 15.6.2 [class.base.init] paragraph 8 as follows:

   ...**An attempt to initialize more than one non-static data member of a union renders the program ill-formed.** After the call to a constructor for class `X` has completed...

---

710. Data races during construction

Section: 15.7 [class.cdtor] **Status:** CD2 **Submitter:** Jeffrey Yasskin **Date:** 3 May, 2008

[Voted into WP at March, 2010 meeting.]

Consider the following example:

```
struct A {
  A() {
    std::thread(&A::Func, this).detach();
  }
  virtual void Func() {
    printf("In A");
  }
};

struct B : public A {
  virtual void Func() {
    printf("In B");
  }
};

struct C : public B {
  virtual void Func() {
    printf("In C");
  }
};

C c;
```

What is the program allowed to print? Should it be undefined behavior or merely unspecified which of the `Func()`s is called?

There is a related question about which variables `C::Func()` can depend on having been constructed. Unless we want to require the equivalent of at least `memory_order_consume` on the presumed virtual function table pointer, I think the answer is just the members of `A`.

If I instead just have

```
A a;
```

I think the only reasonable behavior is to print `In A`.

Finally, given

```
struct F {
  F() {
    std::thread(&F::Func, this).detach();
  }
  virtual void Func() {
    print("In F");
  }
};

struct G : public F {
};

G g;
```

I can see the behavior being undefined, but I think a lot of people would be confused if it did anything other than print `In F`.

Suggested resolution:

I think the intent here is that an object should not be used in another thread until any non-trivial constructor has been called. One possible way of saying that would be to add a new paragraph at the end of 15.7 [class.cdtor]:

> A constructor for a class with virtual functions or virtual base classes modifies a memory location in the object that is accessed by any access to a virtual function or virtual base class or by a `dynamic_cast`. [*Note:* This implies that access to an object by another thread while it is being constructed often introduces a data race (see 4.7 [intro.multithread]). —*end note*]

Proposed resolution (October, 2009):

Add the following as a new paragraph at the end of 6.8 [basic.life]:

> **In this section, "before" and "after" refer to the "happens before" relation (4.7 [intro.multithread]). [*Note:* Therefore, undefined behavior results if an object that is being constructed in one thread is referenced from a different thread without adequate synchronization. —*end note*]**

---

653. Copy assignment of unions

Section: 15.8 [class.copy] **Status:** CD2 **Submitter:** Jens Maurer **Date:** 3 October 2007

[Voted into WP at July, 2009 meeting.]

How does copy assignment for unions work? For example,

```
union U {
  int a;
  float b;
};

void f() {
  union U u = { 5 };
  union U v;
  v = u;    // what happens here?
}
```

12.3 [class.union] is silent on the issue, therefore it seems that 15.8 [class.copy] applies. There is no special case for unions, thus paragraph 13 (memberwise assignment of subobjects) seems to apply. That would seem to imply these actions in the compiler-generated copy assignment operator:

```
v.a = u.a;
v.b = u.b;
```

And this is just wrong. For example, the lifetime of `v.a` ends once the second assignment reuses the memory of `v.a`.

We should probably prescribe "memcpy" copying for unions (both for the copy constructor and the assignment operator) unless the user provided his own special member function.

Proposed resolution (March, 2008):

1. Change 15.8 [class.copy] paragraph 8 as follows:

   > The implicitly-defined or explicitly-defaulted copy constructor for **a non-union** class `X` performs a memberwise copy of its subobjects...

2. Add a new paragraph after 15.8 [class.copy] paragraph 8:

   > The implicitly-defined or explicitly-defaulted copy constructor for a union `X` where all members have a trivial copy constructor copies the object representation (6.9 [basic.types]) of `X`. [*Note:* The behavior is undefined if `X` is not a trivial type. —*end note*]

3. Change 15.8 [class.copy] paragraph 13 as follows:

   > The implicitly-defined or explicitly-defaulted copy assignment operator for **a non-union** class `X` performs memberwise assignment of its subobjects...

4. Add a new paragraph after 15.8 [class.copy] paragraph 13:

   > The implicitly-defined or explicitly-defaulted copy assignment operator for a union `X` where all members have a trivial copy assignment operator copies the object representation (6.9 [basic.types]) of `X`. [*Note:* The behavior is undefined if `X` is not a trivial type. —*end note*]

Notes from the September, 2008 meeting:

The proposed wording needs to be updated to reflect the changes adopted in papers N2757 and N2762, resolving issue 683, which require "no non-trivial" special member functions instead of "a trivial" function. Also, the notes regarding undefined behavior are incorrect, because the member functions involved are defined as deleted when there are non-trivial members.

Proposed resolution (October, 2008):

1. Change 15.8 [class.copy] paragraph 8 as follows:

   > The implicitly-defined or explicitly-defaulted copy constructor for **a non-union** class `X` performs a memberwise copy of its subobjects...

2. Add a new paragraph following 15.8 [class.copy] paragraph 8:

   > The implicitly-defined or explicitly-defaulted copy constructor for a union `X` copies the object representation (6.9 [basic.types]) of `X`.

3. Change 15.8 [class.copy] paragraph 13 as follows:

The implicitly-defined or explicitly-defaulted copy assignment operator for **a non-union** class `X` performs memberwise assignment of its subobjects...

4. Add a new paragraph following 15.8 [class.copy] paragraph 13:

> The implicitly-defined or explicitly-defaulted copy assignment operator for a union `X` copies the object representation (6.9 [basic.types]) of `X`.

---

667. Trivial special member functions that cannot be implicitly defined

Section: 15.8 [class.copy] **Status:** CD2 **Submitter:** James Widman **Date:** 14 December 2007

[Voted into WP at March, 2010 meeting as part of document N3079.]

Should the following class have a trivial copy assignment operator?

```
struct A {
    int& m;
    A();
    A(const A&);
};
```

15.8 [class.copy] paragraph 11 does not mention whether the presence of reference members (or cv-qualifiers, etc.) should affect triviality. Should it?

One reason why this matters is that implementations have to make the builtin type trait operator `__has_trivial_default_ctor(T)` work so that they can support the type trait template `std::has_trivial_default_constructor`.

Assuming the answer is "yes," it looks like we probably need similar wording for trivial default and trivial copy ctors.

Notes from the February, 2008 meeting:

Deleted special member functions are also not trivial. Resolution of this issue should be coordinated with the concepts proposal.

Notes from the June, 2008 meeting:

It appears that this issue will be resolved by the concepts proposal directly. The issue is in "review" status to check if that is indeed the case in the final version of the proposal.

Additional notes (May, 2009):

Consider the following example:

```
struct Base {
  private:
    ~Base() = default;
};

struct Derived: Base {
};
```

The implicitly-declared destructor of `Derived` is defined as deleted because `Base::~Base()` is inaccessible, but it fulfills the requirements for being trivial. Presumably the `Base` destructor should be non-trivial, either by directly specifying that it is non-trivial or by specifying that it is user-provided. An alternative would be to make it ill-formed to attempt to declare a defaulted non-public special member function.

Any changes to the definition of triviality should be checked against 12 [class] paragraph 6 for any changes needed there to accommodate the new definitions.

Notes from the July, 2009 meeting:

The July, 2009 resolution of issue 906 addresses the example above (with an inaccessible defaulted destructor): a defaulted special member function can only have non-public access if the defaulted definition is outside the class, making it non-trivial. The example as written above would be ill-formed.

Proposed resolution (October, 2009):

1. Change 11.4 [dcl.fct.def] paragraph 9 as follows:

> ...Only special member functions may be explicitly defaulted**. Explicitly-defaulted functions and implicitly-declared functions are collectively called *defaulted* functions**, and the implementation shall ~~define them as if they had~~ **provide** implicit definitions **for them** (15.1 [class.ctor], 15.4 [class.dtor], 15.8 [class.copy])**, which might mean defining them as deleted**. A special member function that would be implicitly defined as deleted may be explicitly defaulted only on its first declaration, in which case it is defined as deleted. A special member function is *user-provided* if it is user-declared and not explicitly defaulted on its first declaration. A user-provided explicitly-defaulted function is defined at the point where it is explicitly defaulted. [*Note:*...

2. Change 15.1 [class.ctor] paragraphs 5-6 as follows:

> A *default* constructor for a class `X` is a constructor of class `X` that can be called without an argument. If there is no user-declared constructor for class `X`, a constructor having no parameters is implicitly declared **as defaulted (11.4 [dcl.fct.def])**.

An implicitly-declared default constructor is an `inline public` member of its class. ~~A default constructor is *trivial* if it is not user-provided (11.4 [dcl.fct.def]) and if:~~

- ~~its class has no virtual functions (13.3 [class.virtual]) and no virtual base classes (13.1 [class.mi]), and~~
- ~~no non-static data member of its class has a *brace-or-equal-initializer*, and~~
- ~~all the direct base classes of its class have trivial default constructors, and~~
- ~~for all the non-static data members of its class that are of class type (or array thereof), each such class has a trivial default constructor.~~

A~~n implicitly-declared~~ **defaulted** default constructor for class `X` is defined as deleted if:

- `X` is a union-like class that has a variant member with a non-trivial default constructor,
- any non-static data member is of reference type,
- any non-static data member of const-qualified type (or array thereof) does not have a user-provided default constructor, or
- any non-static data member or direct or virtual base class has class type `M` (or array thereof) and `M` has no default constructor, or if overload resolution (16.3 [over.match]) as applied to `M`'s default constructor, results in an ambiguity or a function that is deleted or inaccessible from the implicitly-declared default constructor.

A default constructor is trivial if it is neither user-provided nor deleted and if:

- **its class has no virtual functions (13.3 [class.virtual]) and no virtual base classes (13.1 [class.mi]), and**
- **no non-static data member of its class has a *brace-or-equal-initializer*, and**
- **all the direct base classes of its class have trivial default constructors, and**
- **for all the non-static data members of its class that are of class type (or array thereof), each such class has a trivial default constructor.**

Otherwise, the default constructor is *non-trivial*.

A ~~non-user-provided~~ default constructor ~~for a class~~ **that is defaulted and not deleted** is *implicitly defined* when it is used (6.2 [basic.def.odr]) to create an object of its class type (4.5 [intro.object])**, or when it is explicitly defaulted after its first declaration**. The implicitly-defined ~~or explicitly-defaulted~~ default constructor performs the set of initializations of the class that would be performed by a user-written default constructor for that class with no *ctor-initializer* (15.6.2 [class.base.init]) and an empty *compound-statement*. If that user-written default constructor would be ill-formed, the program is ill-formed. If that user-written default constructor would satisfy the requirements of a constexpr constructor (10.1.5 [dcl.constexpr]), the implicitly-defined default constructor is constexpr. Before the ~~non-user-provided~~ **defaulted** default constructor for a class is implicitly defined, all the non-user-provided default constructors for its base classes and its non-static data members shall have been implicitly defined. [*Note:* an implicitly-declared default constructor has an *exception-specification* (18.4 [except.spec]). An explicitly-defaulted definition has no implicit *exception-specification*. —*end note*]

3. Change 15.4 [class.dtor] paragraphs 3-4 as follows:

If a class has no user-declared destructor, a destructor is ~~declared~~ implicitly **declared as defaulted (11.4 [dcl.fct.def])**. An implicitly-declared destructor is an `inline public` member of its class. ~~If the class is a union-like class that has a variant member with a non-trivial destructor, an implicitly-declared destructor is defined as deleted (11.4 [dcl.fct.def]). A destructor is *trivial* if it is not user-provided and if:~~

- ~~the destructor is not `virtual`;~~
- ~~all of the direct base classes of its class have trivial destructors, and~~
- ~~for all of the non-static data members of its class that are of class type (or array thereof), each such class has a trivial destructor.~~

A~~n implicitly-declared~~ **defaulted** destructor for a class `X` is defined as deleted if:

- `X` is a union-like class that has a variant member with a non-trivial destructor,
- any of the non-static data members has class type `M` (or array thereof) and `M` has ~~an~~ **a** deleted destructor or a destructor that is inaccessible from the implicitly-declared destructor, or
- any direct or virtual base class has a deleted destructor or a destructor that is inaccessible from the implicitly-declared destructor.

A destructor is trivial if it is neither user-provided nor deleted and if:

- **the destructor is not `virtual`,**
- **all of the direct base classes of its class have trivial destructors, and**
- **for all of the non-static data members of its class that are of class type (or array thereof), each such class has a trivial destructor.**

Otherwise, the destructor is *non-trivial*.

A ~~non-user-provided~~ destructor **that is defaulted and not defined as deleted** is *implicitly defined* when it is used to destroy an object of its class type (6.7 [basic.stc])**, or when it is explicitly defaulted after its first declaration**. ~~A program is ill-formed if the class for which a destructor is implicitly defined or explicitly defaulted has:~~

- ~~a non-static data member of class type (or array thereof) with an inaccessible destructor, or~~
- ~~a base class with an inaccessible destructor.~~

Before the ~~non-user-provided~~ **defaulted** destructor for a class is implicitly defined, all the ~~non-user-defined~~ **non-user-provided** destructors for its base classes and its non-static data members shall have been implicitly defined. [*Note:* an implicitly-declared destructor has an *exception-specification* (18.4 [except.spec]). An explictly defaulted definition has no implicit *exception-specification*. *—end note*]

4. Change 15.8 [class.copy] paragraphs 4-9 as follows:

If the class definition does not explicitly declare a copy constructor, one is ~~declared *implicitly*~~ **implicitly declared as defaulted (11.4 [dcl.fct.def])**. Thus...

...An implicitly-declared copy constructor is an `inline public` member of its class. A~~n implicitly declared~~ **defaulted** copy constructor for a class `X` is defined as deleted if `X` has: ...

A copy constructor for class `X` is ~~*trivial*~~ **trivial** if it is ~~not~~ **neither** user-provided **nor deleted** ~~(11.4 [dcl.fct.def])~~ and if...

A ~~non-user-provided~~ copy constructor **that is defaulted and not defined as deleted** is *implicitly defined* if it is used to initialize an object of its class type from a copy of an object of its class type or of a class type derived from its class type[116]**, or when it is explicitly defaulted after its first declaration**. [*Note:* the copy constructor is implicitly defined even if the implementation elided its use (15.2 [class.temporary]). *—end note*]

Before the ~~non-user-provided~~ **defaulted** copy constructor for a class is implicitly defined, all non-user-provided copy constructors...

The implicitly-defined ~~or explicitly-defaulted~~ copy constructor for a non-union class `X` performs...

The implicitly-defined ~~or explicitly-defaulted~~ copy constructor for a union `X` copies the object representation (6.9 [basic.types]) of `X`.

5. Change 15.8 [class.copy] paragraphs 11-15 as follows:

If the class definition does not explicitly declare a copy assignment operator, one is ~~declared *implicitly*~~ **implicitly declared as defaulted (11.4 [dcl.fct.def])**...

...A~~n implicitly declared~~ **defaulted** copy assignment operator for class `X` is defined as deleted if `X` has:...

A copy assignment operator for class `X` is *trivial* if it is ~~not~~ **neither** user-provided **nor deleted** and if...

A ~~non-user-provided~~ copy assignment operator **that is defaulted and not defined as deleted** is *implicitly defined* when an object of its class type is assigned a value of its class type or a value of a class type derived from its class type**, or when it is explicitly defaulted after its first declaration**.

Before the ~~non-user-provided~~ **defaulted** copy assignment operator for a class is implicitly defined...

The implicitly-defined ~~or explicitly-defaulted~~ copy assignment operator for a non-union class `X` performs...

It is unspecified whether subobjects representing virtual base classes are assigned more than once by the implicitly-defined ~~or explicitly-defaulted~~ copy assignment operator. [*Example:*...

The implicitly-defined ~~or explicitly-defaulted~~ copy assignment operator for a union `X` copies the object representation (6.9 [basic.types]) of `X`.

---

680. What is a move constructor?

Section: 15.8 [class.copy] **Status:** CD2 **Submitter:** Steve Adamczyk **Date:** 3 March, 2008

N2800 comment US 33

[Voted into the WP at the July, 2009 meeting as part of N2927.]

Although the term "move constructor" appears multiple times in the library clauses and is referenced in the newly-added text for the lambda feature, it is not defined anywhere.

Notes from the June, 2008 meeting:

The only reference to "move constructor" in the core language clauses of the Standard is in 8.1.5 [expr.prim.lambda] paragraph 10; there are no semantic implications of the term. This issue will be addressed by using a function signature instead of the term, thus allowing the library section to provide a definition that is appropriate for its needs.

Proposed resolution (July, 2009)

See document PL22.16/09-0117 = WG21 N2927.

---

887. Move construction of thrown object

Section: 15.8 [class.copy] **Status:** CD2 **Submitter:** Steve Adamczyk **Date:** 6 May, 2009

[Voted into WP at March, 2010 meeting.]

15.8 [class.copy] paragraph 16 details the conditions under which a thrown object can be moved instead of copied. However, the optimization as currently described is unsafe. Consider the following example:

```
void f() {
    X x;
    try {
        throw x;
    } catch (...) {
    }
    // x may have been moved from but can still be accessed here
}
```

When the operation is a throw, as opposed to a return, there must be a restriction that the object potentially being moved be defined within the innermost enclosing try block.

Notes from the July, 2009 meeting:

It is not clear how important this optimization is in the context of `throw`: how often is a large object with substantial copying overhead thrown? Also, throwing an exception is already a heavyweight operation, so presumably moving instead of copying an object would not make much difference.

Proposed resolution (October, 2009):

Change 15.8 [class.copy] paragraph 17 second bullet as follows:

- in a *throw-expression*, when the operand is the name of a non-volatile automatic object **whose scope does not extend beyond the end of the innermost enclosing *try-block* (if there is one)**, the copy operation from the operand to the exception object (18.1 [except.throw]) can be omitted by constructing the automatic object directly into the exception object

---

910. Move constructors and implicitly-declared copy constructors

Section: 15.8 [class.copy] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 4 June, 2009

[Voted into WP at March, 2010 meeting as document N3053.]

A constructor of the form `T::T(T&&)` is a candidate function for copy construction; however, the declaration of such a constructor does not inhibit the implicit declaration and definition of a copy constructor. This can lead to surprising results. We should consider suppressing the implicit copy constructor if a move constructor is declared.

---

999. "Implicit" or "implied" object argument/parameter?

Section: 16.3 [over.match] **Status:** CD2 **Submitter:** James Widman **Date:** 20 November, 2009

[Voted into WP at March, 2010 meeting.]

The terminology used to refer to the parameter for `this` and its corresponding argument is inconsistent, sometimes using "implied" and sometimes "implicit." It would be easier to search the text of the Standard if this usage were made regular.

Proposed resolution (February, 2010):

1. Change the index to refer to "implicit object parameter" and "implied object argument" instead of the current permutations of these terms.

2. Change 16.3 [over.match] paragraph 1 as follows:

   ...how well (for non-static member functions) the object matches the ~~implied~~ **implicit** object parameter...

3. Change 16.3.1 [over.match.funcs] paragraph 4 as follows:

   ...For conversion functions, the function is considered to be a member of the class of the ~~implicit~~ **implied** object argument for the purpose of defining the type of the implicit object parameter...

4. Change the footnote in 16.3.3 [over.match.best] paragraph 1 bullet 1 as follows:

    - [*Footnote:* If a function is a static member function, this definition means that the first argument, the implied object ~~parameter~~ **argument**, has no effect in the determination of whether the function is better or worse than any other function. —*end footnote*]

---

704. To which *postfix-expression*s does overload resolution apply?

Section: 16.3.1.1 [over.match.call] **Status:** CD2 **Submitter:** Jens Maurer **Date:** 29 July, 2008

[Voted into WP at October, 2009 meeting.]

There are several problems with the phrasing of 16.3.1.1 [over.match.call] paragraphs 1 and 3. Paragraph 1 reads,

> Recall from 8.2.2 [expr.call], that a *function call* is a *postfix-expression*, possibly nested arbitrarily deep in parentheses, followed by an optional *expression-list* enclosed in parentheses:
>
> (... ($_{opt}$ *postfix-expression*) ...) $_{opt}$ (*expression-list*$_{opt}$)
>
> Overload resolution is required if the *postfix-expression* is the name of a function, a function template (17.6.6 [temp.fct]), an object of class type, or a set of pointers-to-function.

Aside from the fact that directly addressing the reader ("Recall that...") is stylistically incongruous with the rest of the Standard, as well as the fact that 8.2.2 [expr.call] doesn't mention parentheses at all, this wording does not cover member function calls: a member access expression isn't "the name" of anything. This should perhaps be reworded to refer to being either an *id-expression* or the *id-expression* in a member access expression. This could be either by using two lines in the "of the form" citation or in the discussion following the syntax reference.

In addition, paragraph 3 refers to "a *postfix-expression* of the form `&F`," which is an oxymoron: `&F` is a *unary-expression*, not a *postfix-expression*. One possibility would be to explicitly include the parentheses needed in this case, i.e., "a *postfix-expression* of the form `(&F)`..."

Proposed resolution (September, 2009):

Replace the entirety of 16.3.1.1 [over.match.call] with the following two paragraphs:

> In a function call (8.2.2 [expr.call])
>
> *postfix-expression* (*expression-list*$_{opt}$)
>
> if the *postfix-expression* denotes a set of overloaded functions and/or function templates, overload resolution is applied as specified in 16.3.1.1.1 [over.call.func]. If the *postfix-expression* denotes an object of class type, overload resolution is applied as specified in 16.3.1.1.2 [over.call.object].
>
> If the *postfix-expression* denotes the address of a set of overloaded functions and/or function templates, overload resolution is applied using that set as described above. If the function selected by overload resolution is a non-static member function, the program is ill-formed. [*Note:* The resolution of the address of an overload set in other contexts is described in 16.4 [over.over]. —*end note*]

---

604. Argument list for overload resolution in copy-initialization

Section: 16.3.1.3 [over.match.ctor] **Status:** CD2 **Submitter:** Dawn Perchik **Date:** 4 November 2006

[Voted into WP at October, 2009 meeting.]

According to 16.3.1.3 [over.match.ctor],

> When objects of class type are direct-initialized (11.6 [dcl.init]), or copy-initialized from an expression of the same or a derived class type (11.6 [dcl.init])... [the] argument list is the *expression-list* within the parentheses of the initializer.

However, in copy initialization (using the "=" notation), there need be no parentheses. What is the argument list in that case?

Proposed resolution (June, 2009):

Change 16.3.1.3 [over.match.ctor] paragraph 1 as follows:

> ...The argument list is the *expression-list* **or *assignment-expression*** ~~within the parentheses~~ of the ~~initializer~~ ***initializer***.

---

899. Explicit conversion functions in direct class initialization

Section: 16.3.1.4 [over.match.copy] **Status:** CD2 **Submitter:** Jason Merrill **Date:** 13 May, 2009

[Voted into WP at March, 2010 meeting.]

Consider the following example:

```
struct C { };

struct A {
   explicit operator int() const;
   explicit operator C() const;
};

struct B {
   int i;
   B(const A& a): i(a) { }
};

int main() {
   A a;
   int i = a;
   int j(a);
   C c = a;
   C c2(a);
}
```

It's clear that the `B` constructor and the declaration of `j` are well-formed and the declarations of `i` and `c` are ill-formed. But what about the declaration of `c2`? This is supposed to work, but it doesn't under the current wording.

`C c2(a)` is direct-initialization of a class, so constructors are considered. The only possible candidate is the default copy constructor. So we look for a conversion from `A` to `const C&`. There is a conversion operator to `C`, but it is explicit and we are now performing copy-initialization of a reference temporary, so it is not a candidate, and the declaration of `c2` is ill-formed.

Proposed resolution (October, 2009):

Change 16.3.1.4 [over.match.copy] paragraph 1 second bullet as follows:

- When the type of the initializer expression is a class type "*cv* `S`", the non-explicit conversion functions of `S` and its base classes are considered. **When initializing a temporary to be bound to the first parameter of a copy constructor (15.8 [class.copy]) called with a single argument in the context of direct-initialization, explicit conversion functions are also considered.** Those that are not hidden within `S` and yield a type whose cv-unqualified version is the same type as `T` or is a derived class thereof are candidate functions. Conversion functions that return "reference to `X`" return lvalues or rvalues, depending on the type of reference, of type `X` and are therefore considered to yield `X` for this process of selecting candidate functions.

---

641. Overload resolution and conversion-to-same-type operators

Section: 16.3.2 [over.match.viable] **Status:** CD2 **Submitter:** Nathan Sidwell **Date:** 2 Aug 2007

[Voted into the WP at the March, 2009 meeting.]

15.3.2 [class.conv.fct] paragraph 1 says,

> A conversion function is never used to convert a (possibly cv-qualified) object to the (possibly cv-qualified) same object type (or a reference to it), to a (possibly cv-qualified) base class of that type (or a reference to it), or to (possibly cv-qualified) void.

At what point is this enforced, and how is it enforced?

1. Does such a user-declared conversion operator participate in overload resolution? Or is it never entered into the overload set?
2. If it does participate in overload resolution, what happens if it is selected? Is the program ill-formed (and diagnostic required), or is it silently ignored? The above wording doesn't really make it clear.

Consider this test case:

```
struct abc;

struct xyz {
   xyz();

   xyz(xyz &);

   operator xyz& (); // #1
   operator abc& (); // #2
};

struct abc : xyz {};

void foo(xyz &);

void bar() {
        foo (xyz ());
}
```

If such conversion functions are part of the overload set, #1 is a better conversion than #2 to convert the temporary xyz object to a non-const reference required for foo's operand. If such conversion functions are not part of the overload set, then #2 would be selected, and AFAICT the program would be well formed.

If the conversion functions are not part of the overload set, then it would seem one cannot take their address. For instance, adding the following line to the above test case would find no suitable function:

```
xyz &(xyz::*ptr) () = &xyz::operator xyz &;
```

Notes from the October, 2007 meeting:

The intent of 15.3.2 [class.conv.fct] paragraph 1 is that overload resolution not be attempted at all for the listed cases; that is, if the target type is `void`, the object's type, or a base of the object's type, the conversion is done directly without considering any conversion functions. Consequently, the questions about whether the conversion function is part of the overload set or not are moot. The wording will be changed to make this clearer.

Proposed Resolution (October, 2007):

Change the footnote in 15.3.2 [class.conv.fct] paragraph 1 as follows:

> A conversion function is never used to convert a (possibly cv-qualified) object to the (possibly cv-qualified) same object type (or a reference to it), to a (possibly cv-qualified) base class of that type (or a reference to it), or to (possibly cv-qualified) void. [*Footnote:* **These conversions are considered as standard conversions for the purposes of overload resolution (16.3.3.1 [over.best.ics], 16.3.3.1.4 [over.ics.ref]) and therefore initialization (11.6 [dcl.init]) and explicit casts (8.2.9 [expr.static.cast]). A conversion to `void` does not invoke any conversion function (8.2.9 [expr.static.cast]).** Even though never directly called to perform a conversion, such conversion functions can be declared and can potentially be reached through a call to a virtual conversion function in a base class —*end footnote*]

Additional note (March, 2008):

A slight change to the example above indicates that there is a need for a normative change as well as the clarification of the rationale in the October, 2007 proposed resolution. If the declaration of `foo` were changed to

```
void foo(const xyz&);
```

with the current wording, the call `foo(xyz())` would be interpreted as `foo(xyz().operator abc&())` instead of binding the parameter directly to the rvalue, which is clearly wrong.

Proposed resolution (March, 2008):

1. Change the footnote in 15.3.2 [class.conv.fct] paragraph 1 as described in the October, 2007 proposed resolution.
2. Change 11.6.3 [dcl.init.ref] paragraph 5 as follows:

> A reference to type "*cv1* `T1`" is initialized by an expression of type "*cv2* `T2`" as follows:
>
> - If the initializer expression
>
>   - is an lvalue (but is not a bit-field), and "*cv1* `T1`" is reference-compatible with "*cv2* `T2`," or
>
>   - has a class type (i.e., `T2` is a class type)**, where `T1` is not reference-related to `T2`,** and can be implicitly converted to an lvalue of type "*cv3* `T3`," where "*cv1* `T1`" is reference-compatible with "*cv3* `T3`" [*Footnote:* This requires a conversion function (15.3.2 [class.conv.fct]) returning a reference type. —*end footnote*] (this conversion is selected by enumerating the applicable conversion functions (16.3.1.6 [over.match.ref]) and choosing the best one through overload resolution (16.3 [over.match])),
>
> then...

[Drafting note: this resolution makes the example in the issue description ill-formed.]

---

877. Viable functions and binding references to rvalues

Section: 16.3.2 [over.match.viable] **Status:** CD2 **Submitter:** Daniel Krügler **Date:** 23 April, 2009

[Voted into WP at October, 2009 meeting.]

16.3.2 [over.match.viable] paragraph 3 says,

> If the parameter has reference type, the implicit conversion sequence includes the operation of binding the reference, and the fact that a reference to non-const cannot be bound to an rvalue can affect the viability of the function (see 16.3.3.1.4 [over.ics.ref]).

This should say "lvalue reference to non-const," as is correctly stated in 16.3.3.1.4 [over.ics.ref] paragraph 3.

Proposed resolution (July, 2009):

Change 16.3.2 [over.match.viable] paragraph 3 as follows:

> If the parameter has reference type, the implicit conversion sequence includes the operation of binding the reference, and the fact that ~~a~~ **an lvalue** reference to non-const cannot be bound to an rvalue can affect the viability of the function (see 16.3.3.1.4 [over.ics.ref]).

---

495. Overload resolution with template and non-template conversion functions

Section: 16.3.3 [over.match.best] **Status:** CD2 **Submitter:** Nathan Sidwell **Date:** 20 Dec 2004

[Voted into WP at July, 2009 meeting.]

The overload resolution rules for ranking a template against a non-template function differ for conversion functions in a surprising way. 16.3.3 [over.match.best] lists four checks, the last three concern this report. For the non-conversion operator case, checks 2 and 3 are applicable, whereas for the conversion operator case checks 3 and 4 are applicable. Checks 2 and 4 concern the ranking of argument and return value conversion sequences respectively. Check 3 concerns only the templatedness of the functions being ranked, and will prefer a non-template to a template. Notice that this check happens after argument conversion sequence ranking, but *before* return value conversion sequence ranking. This has the effect of always selecting a non-template conversion operator, as the following example shows:

```
struct C
{
  inline operator int () { return 1; }
  template <class T> inline operator T () { return 0; }
};

inline long f (long x) { return x; }

int
main (int argc, char *argv[])
{
  return f (C ());
}
```

The non-templated `C::operator int` function will be selected, rather than the apparently better `C::operator long<long>` instantiation. This is a surprise, and resulted in a bug report where the user expected the template to be selected. In addition some C++ compilers have implemented the overload ranking as if checks 3 and 4 were transposed.

Is this ordering accidental, or is there a rationale?

Notes from the April, 2005 meeting:

The CWG agreed that the template/non-template distinction should be the final tie-breaker.

Proposed resolution (March, 2007):

In the second bulleted list of 16.3.3 [over.match.best] paragraph 1, move the second and third bullets to the end of the list, to read as follows:

- for some argument *i*, ICS*i*(`F1`) is a better conversion sequence than ICS*i*(`F2`), or, if not that,
- the context is an initialization by user-defined conversion (see 11.6 [dcl.init], 16.3.1.5 [over.match.conv], and 16.3.1.6 [over.match.ref]) and the standard conversion sequence from the return type of `F1` to the destination type (i.e., the type of the entity being initialized) is a better conversion sequence than the standard conversion sequence from the return type of `F2` to the destination type, [*Example:* ... —*end example*] or, if not that,
- `F1` is a non-template function and `F2` is a function template specialization, or, if not that,
- `F1` and `F2` are function template specializations, and the function template for `F1` is more specialized than the template for `F2` according to the partial ordering rules described in 17.6.6.2 [temp.func.order].

---

978. Incorrect specification for copy initialization

Section: 16.3.3.1 [over.best.ics] **Status:** CD2 **Submitter:** Daniel Krügler **Date:** 5 October, 2009

[Voted into WP at March, 2010 meeting.]

16.3.3.1 [over.best.ics] paragraph 4 says,

> However, when considering the argument of a user-defined conversion function that is a candidate by 16.3.1.3 [over.match.ctor] when invoked for the copying of the temporary in the second step of a class copy-initialization, by 16.3.1.7 [over.match.list] when passing the initializer list as a single argument or when the initializer list has exactly one element and a conversion to some class `X` or reference to (possibly cv-qualified) `X` is considered for the first parameter of a constructor of `X`, or by 16.3.1.4 [over.match.copy], 16.3.1.5 [over.match.conv], or 16.3.1.6 [over.match.ref] in all cases, only standard conversion sequences and ellipsis conversion sequences are allowed.

This is not quite right, as this applies to constructor arguments, not just arguments of user-defined conversion functions. Furthermore, the word "allowed" might be better replaced by something like,

> considered (in particular, for the purposes of determining whether the candidate function (that is either a constructor or a conversion function) is viable)

Proposed resolution (October, 2009):

Change 16.3.3.1 [over.best.ics] paragraph 4 as follows:

> However, when considering the argument of a **constructor or** user-defined conversion function that is a candidate by 16.3.1.3 [over.match.ctor] when invoked for the copying of the temporary in the second step of a class copy-initialization, by 16.3.1.7 [over.match.list] when passing the initializer list as a single argument or when the initializer list has exactly one element and a conversion to some class `X` or reference to (possibly cv-qualified) `X` is considered for the first parameter of a constructor of `X`, or by 16.3.1.4 [over.match.copy], 16.3.1.5 [over.match.conv], or 16.3.1.6 [over.match.ref] in all cases, only standard conversion sequences and ellipsis conversion sequences are ~~allowed~~ **considered**.

---

953. Rvalue references and function viability

Section: 16.3.3.1.4 [over.ics.ref] **Status:** CD2 **Submitter:** Mike Miller **Date:** 18 August, 2009

[Voted into WP at March, 2010 meeting.]

According to 16.3.3.1.4 [over.ics.ref] paragraphs 3-4,

> A standard conversion sequence cannot be formed if it requires binding an lvalue reference to non-const to an rvalue (except when binding an implicit object parameter; see the special rules for that case in 16.3.1 [over.match.funcs]). [*Note:* this means, for example, that a candidate function cannot be a viable function if it has a non-`const` lvalue reference parameter (other than the implicit object parameter) and the corresponding argument is a temporary or would require one to be created to initialize the lvalue reference (see 11.6.3 [dcl.init.ref]). —*end note*]
>
> Other restrictions on binding a reference to a particular argument that are not based on the types of the reference and the argument do not affect the formation of a standard conversion sequence, however.

Because this section does not mention attempting to bind an rvalue reference to an lvalue, such a "conversion sequence" might be selected as best and result in an ill-formed program. It should, instead, be treated like trying to bind an lvalue reference to non-const to an rvalue, making the function non-viable.

Proposed resolution (November, 2009):

Change 16.3.3.1.4 [over.ics.ref] paragraph 3 as follows:

> ~~A~~ **Except for an implicit object parameter, for which see 16.3.1 [over.match.funcs], a** standard conversion sequence cannot be formed if it requires binding an lvalue reference to non-`const` to an rvalue ~~(except when binding an implicit object parameter; see the special rules for that case in 16.3.1 [over.match.funcs])~~ **or binding an rvalue reference to an lvalue**. [*Note:* this means, for example, that a candidate function cannot be a viable function if it has a non-`const` lvalue reference parameter (other than the implicit object parameter) and the corresponding argument is a temporary or would require one to be created to initialize the lvalue reference (see 11.6.3 [dcl.init.ref]). —*end note*]

---

702. Preferring conversion to `std::initializer_list`

Section: 16.3.3.2 [over.ics.rank] **Status:** CD2 **Submitter:** Jason Merrill **Date:** 2 July, 2008

[Voted into WP at July, 2009 meeting.]

We need another bullet in 16.3.3.2 [over.ics.rank], along the lines of:

- List-initialization sequence `L1` is a better conversion sequence than list-initialization sequence `L2` if `L1` converts to `std::initializer_list<X>` for some `X` and `L2` does not.

This is necessary to make the following example work:

```
#include <initializer_list>

struct string {
  string (const char *) {}
  template <class Iter> string (Iter, Iter);
};

template <class T, class U>
struct pair {
  pair (T t, U u) {}
};

template<class T, class U>
struct map {
  void insert (pair<T,U>);
  void insert (std::initializer_list<pair<T,U> >) {}
};

int main() {
  map<string,string> m;
  m.insert({ {"this","that"}, {"me","you"} });
}
```

Proposed resolution (March, 2009):

Add a new top-level bullet at the end of the current list in 16.3.3.2 [over.ics.rank] paragraph 3:

- List-initialization sequence `L1` is a better conversion sequence than list-initialization sequence `L2` if `L1` converts to `std::initializer_list<X>` for some `X` and `L2` does not.

---

961. Overload resolution and conversion of `std::nullptr_t` to `bool`

Section: 16.3.3.2 [over.ics.rank] **Status:** CD2 **Submitter:** Mike Miller **Date:** 2 September, 2009

[Voted into WP at March, 2010 meeting.]

Conversion of a pointer or pointer to member to `bool` is given special treatment as a tiebreaker in overload resolution in 16.3.3.2 [over.ics.rank] paragraph 4, bullet 1:

- A conversion that is not a conversion of a pointer, or pointer to member, to `bool` is better than another conversion that is such a conversion.

It would be reasonable to expect a similar provision to apply to conversions of `std::nullptr_t` to `bool`.

Proposed resolution (October, 2009):

Change 16.3.3.2 [over.ics.rank] paragraph 4 bullet 1 as follows:

- A conversion that ~~is not a conversion of~~ **does not convert** a pointer, ~~or~~ **a** pointer to member, **or** `std::nullptr_t` to `bool` is better than ~~another conversion that is such a conversion~~ **one that does**.

---

935. Missing overloads for character types for user-defined literals

Section: 16.5.8 [over.literal] **Status:** CD2 **Submitter:** Alisdair Meredith **Date:** 9 July, 2009

[Voted into WP at March, 2010 meeting.]

The list of overloads for user-defined literal operators given in 16.5.8 [over.literal] paragraph 3 should include signatures for `char`, `wchar_t`, `char16_t`, and `char32_t`.

Proposed resolution (November, 2009):

Change 16.5.8 [over.literal] paragraph 3 as follows:

> The declaration of a literal operator shall have a *parameter-declaration-clause* equivalent to one of the following:

```
const char*
unsigned long long int
long double
char
wchar_t
char16_t
char32_t
const char*, std::size_t
const wchar_t*, std::size_t
const char16_t*, std::size_t
const char32_t*, std::size_t
```

---

749. References to function types with a *cv-qualifier* or *ref-qualifier*

Section: 16.6 [over.built] **Status:** CD2 **Submitter:** Alberto Ganesh Barbati **Date:** 9 December, 2008

[Voted into WP at July, 2009 meeting.]

16.6 [over.built] paragraph 7 posits the existence of built-in candidate `operator*` functions "for every function type *T*." However, only non-static member function types can contain a *cv-qualifier* or *ref-qualifier* (11.3.5 [dcl.fct] paragraph 7), and a reference to such a type cannot be initialized (8.2.5 [expr.ref] paragraph 4, bullet 3, sub-bullet 2). (See also _N2914_.14.10.4 [concept.support] paragraph 10, which disallows references to function types with *cv-qualifier*s but is silent on *ref-qualifier*s.)

Proposed resolution (March, 2009):

1. Change 16.6 [over.built] paragraph 7 as follows:

> For every function type *T* **that does not have cv-qualifiers or a *ref-qualifier***, there exist candidate operator functions of the form

```
T& operator*(T*);
```

2. Change _N2914_.14.10.4 [concept.support] paragraph 7 as follows:

   Requires: for every type `T` that is an object type, a function type that does not have cv-qualifiers **or a *ref-qualifier***, or *cv* `void`, a concept map `PointeeType<T>` is implicitly defined in namespace `std`.

3. Change _N2914_.14.10.4 [concept.support] paragraph 11 as follows:

   Requires: for every type `T` that is an object type, a function type that does not have cv-qualifiers **or a *ref-qualifier***, or a reference type, a concept map `ReferentType<T>` is implicitly defined in namespace `std`.

---

879. Missing built-in comparison operators for pointer types

Section: 16.6 [over.built] **Status:** CD2 **Submitter:** Daniel Krügler **Date:** 25 April, 2009

[Voted into WP at October, 2009 meeting.]

16.6 [over.built] paragraph 15 restricts the built-in comparison operators to

> every *T*, where *T* is an enumeration type or pointer to effective object type

This omits both pointers to function types and pointers to void.

Proposed resolution (July, 2009):

1. Add a new paragraph following 8.9 [expr.rel] paragraph 2:

   Pointers to `void` (after pointer conversions) can be compared, with a result defined as follows: If both pointers represent the same address or are both the null pointer value, the result is `true` if the operator is `<=` or `>=` and `false` otherwise; otherwise the result is unspecified.

2. Change 8.10 [expr.eq] paragraph 1 as follows:

   ...Pointers ~~to objects or functions~~ of the same type (after pointer conversions) can be compared for equality...

3. Change 16.6 [over.built] paragraph 15 as follows:

   For every `T`, where `T` is an enumeration type ~~or~~**, a** pointer ~~to effective object~~ type, **or** `std::nullptr_t`**,** there exist candidate operator functions of the form...

---

880. Built-in conditional operator for scoped enumerations

Section: 16.6 [over.built] **Status:** CD2 **Submitter:** Daniel Krügler **Date:** 25 April, 2009

[Voted into WP at March, 2010 meeting.]

16.6 [over.built] paragraphs 24-25 describe the imaginary built-in conditional operator functions. However, neither paragraph 24 (promoted arithmetic types) nor 25 (pointer and pointer-to-member types) covers scoped enumerations, whose values should be usable in conditional expressions.

(See also issue 835.)

Proposed resolution (October, 2009):

Change 16.6 [over.built] paragraph 25 as follows:

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

---

820. Deprecation of `export`

Section: 17 [temp] **Status:** CD2 **Submitter:** UK **Date:** 3 March, 2009

N2800 comment UK 115

[Voted into WP at March, 2010 meeting as document N3065.]

Exported templates were a great idea that is generally understood to have failed. In the decade since the standard was adopted, only one implementation has appeared. No current vendors appear interested in creating another. We tentatively suggest this makes the feature ripe for deprecation. Our main concern with deprecation is that it might turn out that exported constrained templates become an important compile-time optimization, as the constraints would be checked once in the exported definition and not in each translation unit consuming the exported declarations.

Notes from the March, 2010 meeting:

It was decided to remove `export` altogether, rather than deprecating it.

---

840. Rvalue references as nontype template parameters

Section: 17.1 [temp.param] **Status:** CD2 **Submitter:** Steve Adamczyk **Date:** 13 March, 2009

[Voted into WP at October, 2009 meeting.]

Nontype template parameters are currently allowed to have rvalue reference type (17.1 [temp.param] paragraph 4 bullet 3 just says "reference," not "lvalue reference"). However, with the change of N2844 voted in (which prohibits rvalue references from binding to lvalues), I can't think of any way to specify a valid template argument for a parameter of rvalue reference type. If that's the case, should we restrict nontype template parameters to lvalue reference types?

Proposed resolution (July, 2009):

Change 17.1 [temp.param] paragraph 4, bullet 3 as follows:

- **lvalue** reference to object or **lvalue** reference to function,

---

823. Literal types with constexpr conversions as non-type template arguments

Section: 17.3.2 [temp.arg.nontype] **Status:** CD2 **Submitter:** FR **Date:** 3 March, 2009

N2800 comment FR 29

[Voted into WP at March, 2010 meeting.]

8.20 [expr.const] permits literal types with a `constexpr` conversion function to an integral type to be used in an integral constant expression. However, such conversions are not listed in 17.3.2 [temp.arg.nontype] paragraph 5 bullet 1 among the conversions applied to *template-argument*s for a non-type *template-parameter* of integral or enumeration type.

Notes from the March, 2009 meeting:

The original national body comment suggested allowing any literal type as a non-type template argument. The CWG was not in favor of this change, but in the course of discussing the suggestion discovered the problem with *template-parameter*s of integral and enumeration type.

Proposed resolution (October, 2009):

Change 17.3.2 [temp.arg.nontype] paragraph 1 bullet 1 as follows:

- an integral constant expression **(including a constant expression of literal class type that can be used as an integral constant expression as described in 8.20 [expr.const])**; or

---

744. Matching template arguments with template template parameters with parameter packs

Section: 17.3.3 [temp.arg.template] **Status:** CD2 **Submitter:** Faisal Vali **Date:** 2 November, 2008

[Voted into WP at March, 2010 meeting.]

According to 17.3.3 [temp.arg.template] paragraph 3,

> A *template-argument* matches a template *template-parameter* (call it `P`) when each of the template parameters in the *template-parameter-list* of the *template-argument*'s corresponding class template or template alias (call it `A`) matches the corresponding template parameter in the *template-parameter-list* of `P`. When `P`'s *template-parameter-list* contains a template parameter pack (17.6.3 [temp.variadic]), the template parameter pack will match zero or more template parameters or template parameter packs in the *template-parameter-list* of `A` with the same type and form as the template parameter pack in `P` (ignoring whether those template parameters are template parameter packs).

The immediately-preceding example, however, assumes that a parameter pack in the parameter will match only a parameter pack in the argument:

```
template<class T> class A { /* ... */ };
template<class T, class U = T> class B { /* ... */ };
template<class ... Types> class C { /* ... */ };

template<template<class ...> class Q> class Y { /* ... */ };

Y<A> ya;  // ill-formed: a template parameter pack does not match a template parameter
Y<B> yb;  // ill-formed: a template parameter pack does not match a template parameter
Y<C> yc;  // OK
```

Proposed resolution (February, 2010):

Change the final three lines of the second example in 17.3.3 [temp.arg.template] paragraph 2 as follows:

```
Y<A> ya;  // ~~ill-formed: a template parameter pack does not match a template parameter~~ OK
Y<B> yb;  // ~~ill-formed: a template parameter pack does not match a template parameter~~ OK
Y<C> yc;  // OK
```

---

408. sizeof applied to unknown-bound array static data member of template

Section: 17.6.1.3 [temp.static] **Status:** CD2 **Submitter:** Nathan Myers **Date:** 14 Apr 2003

[Voted into WP at March, 2010 meeting.]

Is this allowed?

```
template<typename T>
  struct X
  {
      static int s[];
      int c;
  };

template<typename T>
  int X<T>::s[sizeof(X<T>)];

int* p = X<char>::s;
```

I have a compiler claiming that, for the purpose of sizeof(), X<T> is an incomplete type, when it tries to instantiate X<T>::s. It seems to me that X<char> should be considered complete enough for sizeof even though the size of s isn't known yet.

John Spicer: This is a problematic construct that is currently allowed but which I think should be disallowed.

I tried this with a number of compilers. None of which did the right thing. The EDG front end accepts it, but gives X<...>::s the wrong size.

It appears that most compilers evaluate the "declaration" part of the static data member definition only once when the definition is processed. The initializer (if any) is evaluated for each instantiation.

This problem is solvable, and if it were the only issue with incomplete arrays as template static data members, then it would make sense to solve it, but there are other problems.

The first problem is that the size of the static data member is only known if a template definition of the static data member is present. This is weird to start with, but it also means that sizes would not be available in general for exported templates.

The second problem concerns the rules for specialization. An explicit specialization for a template instance can be provided up until the point that a use is made that would cause an implicit instantiation. A reference like "sizeof(X<char>::s)" is not currently a reference that would cause an implicit instantiation of X<char>::s. This means you could use such a sizeof and later specialize the static data member with a different size, meaning the earlier sizeof gave the wrong result. We could, of course, change the "use" rules, but I'd rather see us require that static data members that are arrays have a size specified in the class or have a size based on their initializer.

Notes from the October 2003 meeting:

The example provided is valid according to the current standard. A static data member must be instantiated (including the processing of its initializer, if any) if there is any reference to it. The compiler need not, however, put out a definition in that translation unit. The standard doesn't really have a concept of a "partial instantiation" for a static data member, and although we considered adding that, we decided that to get all the size information that seems to be available one needs a full instantiation in any case, so there's no need for the concept of a partial instantiation.

Note (June, 2006):

Mark Mitchell suggested the following example:

```
template <int> void g();

template <typename T>
struct S {
  static int i[];
  void f();
};

template <typename T>
int S<T>::i[] = { 1 };

template <typename T>
void S<T>::f() {
```

```
  g<sizeof (i) / sizeof (int)>();
}

template <typename T>
int S<int>::i[] = { 1, 2 };
```

Which `g` is called from `S<int>::f()`?

If the program is valid, then surely one would expect `g<2>` to be called.

If the program is valid, does `S<T>::i` have a non-dependent type in `S<T>::f`? If so, is it incomplete, or is it `int[1]`? (Here, `int[1]` would be surprising, since `S<int>::i` actually has type `int[2]`.)

If the program is invalid, why?

For a simpler example, consider:

```
template <typename T>
struct S {
  static int i[];
  const int N = sizeof (i);
};
```

This is only valid if the type of `i` is dependent, meaning that the `sizeof` expression isn't evaluated until the class is instantiated.

Proposed resolution (February, 2010):

1. Add the following as a new paragraph following 17.6.1.3 [temp.static] paragraph 1:

   **An explicit specialization of a static data member declared as an array of unknown bound can have a different bound from its definition, if any. [*Example:*

   ```
   template<class T> struct A {
     static int i[];
   };
   template<class T> int A<T>::i[4];    // 4 elements
   template<> int A<int>::i[] = { 1 };  // 1 element, OK
   ```

   —end example]

2. Change 17.7.2.2 [temp.dep.expr] paragraph 3 as follows:

   An *id-expression* is type-dependent if it contains:

   - an *identifier* that was declared with a dependent type,
   - a *template-id* that is dependent,
   - a *conversion-function-id* that specifies a dependent type, **or**
   - a *nested-name-specifier* or a *qualified-id* that names a member of an unknown specialization**;**

   or if it names a static data member of the current instantiation that has type "array of unknown bound of `T`" for some `T` (17.6.1.3 [temp.static]). Expressions of the following forms are type-dependent only if...

---

638. Explicit specialization and friendship

Section: 17.6.4 [temp.friend] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 6 July 2007

[Voted into WP at March, 2010 meeting.]

Is this code well-formed?

```
template <typename T> struct A {
    struct B;
};

class C {
    template <typename T> friend struct A<T>::B;
    static int bar;
};

template <> struct A<char> {
    struct B {
        int f() {
            return C::bar;   // Is A<char>::B a friend of C?
        }
    };
};
```

According to 17.6.4 [temp.friend] paragraph 5,

> A member of a class template may be declared to be a friend of a non-template class. In this case, the corresponding member of every specialization of the class template is a friend of the class granting friendship.

This would tend to indicate that the example is well-formed. However, technically `A<char>::B` does not "correspond to" the same-named member of the class template: 17.8.3 [temp.expl.spec] paragraph 4 says,

> The definition of an explicitly specialized class is unrelated to the definition of a generated specialization. That is, its members need not have the same names, types, etc. as the members of a generated specialization.

In other words, there are no "corresponding members" in an explicit specialization.

Is this the outcome we want for examples like the preceding? There is diversity among implementations on this question, with some accepting the example and others rejecting it as an access violation.

Notes from the July, 2009 meeting:

The consensus of the CWG was to allow the correspondence of similar members in explicit specializations.

Proposed resolution (October, 2009):

Change 17.6.4 [temp.friend] paragraph 5 as follows:

> A member of a class template may be declared to be a friend of a non-template class. In this case, the corresponding member of every specialization of the class template is a friend of the class granting friendship. **For explicit specializations the corresponding member is the member (if any) that has the same name, kind (type, function, class template or function template), template parameters, and signature as the member of the class template instantiation that would otherwise have been generated.** [*Example:*

```
template<class T> struct A {
  struct B { };
  void f();
  struct D {
    void g();
  };
};
template<> struct A<int> {
  struct B { };
  int f();
  struct D {
    void g();
  };
};

class C {
  template<class T> friend struct A<T>::B;    // grants friendship to A<int>::B even though
                                              //   it is not a specialization of A<T>::B
  template<class T> friend void A<T>::f();    // does not grant friendship to A<int>::f()
                                              //   because its return type does not match
  template<class T> friend void A<T>::D::g(); // does not grant friendship to A<int>::D::g()
                                              //   because A<int>::D is not a specialization of A<T>::D
};
```

---

929. What is a template alias?

Section: 17.6.7 [temp.alias] **Status:** CD2 **Submitter:** Alisdair Meredith **Date:** 2 July, 2009

[Voted into WP at October, 2009 meeting.]

Although it is a reasonable assumption that a *template-declaration* in which the *declaration* is an *alias-declaration* declares a template alias, that is not said explicitly in 17.6.7 [temp.alias] nor, apparently, anywhere else.

Proposed resolution (September, 2009):

Change 17.6.7 [temp.alias] paragraph 1 as follows:

> **A *template-declaration* in which the *declaration* is an *alias-declaration* (clause 7) declares the *identifier* to be a *template alias.*** A ~~*template alias* declares~~ **template alias is** a name for a family of types. The name of the template alias is a *template-name*.

---

588. Searching dependent bases of classes local to function templates

Section: 17.7.2 [temp.dep] **Status:** CD2 **Submitter:** James Widman **Date:** 21 June 2006

[Voted into the WP at the March, 2009 meeting.]

17.7.2 [temp.dep] paragraph 3 reads,

> In the definition of a class template or a member of a class template, if a base class of the class template depends on a *template-parameter*, the base class scope is not examined during unqualified name lookup either at the point of definition of the class template or member or during an instantiation of the class template or member.

This wording applies only to definitions of class templates and members of class templates. That would make the following program ill-formed (but it probably should be well-formed):

```
struct B{ void f(int); };

template<class T> struct D: B { };

template<class T> void g() {
   struct B{ void f(); };
   struct A: D<T> {
       B m;
   };
   A a;
   a.m.f(); // Presumably, we want ::g()::B::f(), not ::B::f(int)
}

int main () {
   g<int>();
   return 0;
}
```

I suspect the wording should be something like

> In the definition of a class template **or a class defined (directly or indirectly) within the scope of a class template or function template**, if a base class...

That should also include deeply nested classes in templates, local classes of non-template member functions of member classes of class templates, etc.

Proposed resolution (October, 2006):

Change 17.7.2 [temp.dep] paragraph 3 as follows:

> In the definition of a **class or** class template ~~or a member of a class template~~, if a base class ~~of the class template~~ depends on a *template-parameter*, the base class scope is not examined during unqualified name lookup either at the point of definition of the class template or member or during an instantiation of the class template or member.

---

541. Dependent function types

Section: 17.7.2.2 [temp.dep.expr] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 22 October 2005

[Voted into WP at March, 2010 meeting.]

17.7.2.2 [temp.dep.expr] paragraph 3 says,

> An *id-expression* is type-dependent if it contains:
>
> - an identifier that was declared with a dependent type...

This treatment seems inadequate with regard to *id-expression*s in function calls:

1. According to 17.7.2.1 [temp.dep.type] paragraph 6,

   > A type is dependent if it is
   > - ...
   > - a compound type constructed from any dependent type...

   This would apply to the type of a member function of a class template if any of its parameters are dependent, even if the return type is not dependent. However, there is no need for a call to such a function to be a type-dependent expression because the type of the expression is known at definition time.

2. This wording does not handle the case of overloaded functions, some of which might have dependent types (however defined) and others not.

Notes from the October, 2009 meeting:

The consensus of the CWG was that the first point of the issue is not sufficiently problematic as to require a change.

Proposed resolution (October, 2009):

Change 17.7.2.2 [temp.dep.expr] paragraph 3 as follows:

> An *id-expression* is type-dependent if it contains:
>
> - an *identifier* ~~that was~~ **associated by name lookup with one or more declarations** declared with a dependent type,
> - a *template-id* that is dependent,
> - a *conversion-function-id* that specifies a dependent type, **or**
> - a *nested-name-specifier* or a *qualified-id* that names a member of an unknown specialization.

---

561. Internal linkage functions in dependent name lookup

Section: 17.7.4.2 [temp.dep.candidate] **Status:** CD2 **Submitter:** Joaquín López Muñoz **Date:** 17 February 2006

[Voted into WP at March, 2010 meeting.]

According to 17.7.4.2 [temp.dep.candidate],

> For a function call that depends on a template parameter, if the function name is an *unqualified-id* but not a *template-id*, the candidate functions are found using the usual lookup rules (6.4.1 [basic.lookup.unqual], 6.4.2 [basic.lookup.argdep]) except that:
>
> - For the part of the lookup using unqualified name lookup (6.4.1 [basic.lookup.unqual]), only function declarations with external linkage from the template definition context are found.
> - For the part of the lookup using associated namespaces (6.4.2 [basic.lookup.argdep]), only function declarations with external linkage found in either the template definition context or the template instantiation context are found.

It is not at all clear why a call using a *template-id* would be treated differently from one not using a *template-id*. Furthermore, is it really necessary to exclude internal linkage functions from the lookup? Doesn't the ODR give implementations sufficient latitude to handle this case without another wrinkle on name lookup?

(See also issue 524.)

Notes from the April, 2006 meeting:

The consensus of the group was that *template-id*s should not be treated differently from *unqualified-id*s (although it's not clear how argument-dependent lookup works for *template-id*s), and that internal-linkage functions should be found by the lookup (although they may result in errors if selected by overload resolution).

Note (June, 2006):

Although the notes from the Berlin meeting indicate that argument-dependent lookup for *template-id*s is under-specified in the Standard, further examination indicates that that is not the case: the note in 17.9.1 [temp.arg.explicit] paragraph 8 clearly indicates that argument-dependent lookup is to be performed for *template-id*s, and 6.4.2 [basic.lookup.argdep] paragraph 4 describes the lookup performed:

> When considering an associated namespace, the lookup is the same as the lookup performed when the associated namespace is used as a qualifier (6.4.3.2 [namespace.qual]) except that:
>
> - Any *using-directive*s in the associated namespace are ignored.
> - Any namespace-scope friend functions declared in associated classes are visible within their respective namespaces even if they are not visible during an ordinary lookup (14.3 [class.friend]).

Proposed resolution (October, 2009):

1. Change 17.7.2 [temp.dep] paragraph 1 as follows:

   > In an expression of the form:
   >
   > *postfix-expression* (*expression-list*$_{opt}$)
   >
   > where the *postfix-expression* is an *unqualified-id* ~~but not a *template-id*~~, the *unqualified-id* denotes a *dependent name* if and only if any of the expressions in the *expression-list* is a type-dependent expression (17.7.2.2 [temp.dep.expr])...

2. Change 17.7.4.2 [temp.dep.candidate] paragraph 1 as follows:

   > For a function call that depends on a template parameter, if the function name is an *unqualified-id* ~~but not a *template-id*~~, or if the function is called using operator notation, the candidate functions are found using the usual lookup rules (6.4.1 [basic.lookup.unqual], 6.4.2 [basic.lookup.argdep]) except that:
   >
   > - For the part of the lookup using unqualified name lookup (6.4.1 [basic.lookup.unqual]), only function declarations ~~with external linkage~~ from the template definition context are found.
   > - For the part of the lookup using associated namespaces (6.4.2 [basic.lookup.argdep]), only function declarations ~~with external linkage~~ found in either the template definition context or the template instantiation context are found.

---

969. Explicit instantiation declarations of class template specializations

Section: 17.8.2 [temp.explicit] **Status:** CD2 **Submitter:** Jason Merrill **Date:** 29 December, 2008

[Voted into WP at March, 2010 meeting.]

Consider this example:

```
template <class T> struct A {
   virtual void f() {}
};

extern template struct A<int>;

int main() {
   A<int> a;
   a.f();
}
```

The intent is that the explicit instantiation declaration will suppress any compiler-generated machinery such as a virtual function table or typeinfo data in this translation unit, and that because of 17.8.2 [temp.explicit] paragraph 10,

> An entity that is the subject of an explicit instantiation declaration and that is also used in the translation unit shall be the subject of an explicit instantiation definition somewhere in the program; otherwise the program is ill-formed, no diagnostic required.

the use of `A<int>` in declaring `a` requires an explicit instantiation definition in another translation unit that will provide the requisite compiler-generated data.

The existing wording of 17.8.2 [temp.explicit] does not express this intent clearly enough, however.

Suggested resolution:

1. Change 17.8.2 [temp.explicit] paragraph 7 as follows:

> An explicit instantiation that names a class template specialization is **also** an explicit instantion of the same kind (declaration or definition) of each of its members (not including members inherited from base classes) that has not been previously explicitly specialized in the translation unit containing the explicit instantiation, except as described below.

2. Change 17.8.2 [temp.explicit] paragraph 9 as follows:

> ~~An explicit instantiation declaration that names a class template specialization has no effect on the class template specialization itself (except for perhaps resulting in its implicit instantiation).~~ Except for inline functions **and class template specializations**, other explicit instantiation declarations have the effect of suppressing the implicit instantiation of the entity to which they refer...

Proposed resolution (October, 2009):

Change 17.8.2 [temp.explicit] paragraphs 7-9 as follows:

> An explicit instantiation that names a class template specialization is **also** an explicit instantion of the same kind (declaration or definition) of each of its members (not including members inherited from base classes) that has not been previously explicitly specialized in the translation unit containing the explicit instantiation, except as described below. **[*Note:* In addition, it will typically be an explicit instantiation of certain implementation-dependent data about the class. —*end note*]**
>
> An explicit instantiation definition that names a class template specialization explicitly instantiates the class template specialization and is ~~only~~ an explicit instantiation definition of **only those** members whose definition is visible at the point of instantiation.
>
> ~~An explicit instantiation declaration that names a class template specialization has no effect on the class template specialization itself (except for perhaps resulting in its implicit instantiation).~~ Except for inline functions **and class template specializations**, ~~other~~ explicit instantiation declarations have the effect of suppressing the implicit instantiation of the entity to which they refer...

---

980. Explicit instantiation of a member of a class template

Section: 17.8.2 [temp.explicit] **Status:** CD2 **Submitter:** Doug Gregor **Date:** 14 October, 2009

[Voted into WP at March, 2010 meeting.]

17.8.2 [temp.explicit] paragraph 1 says,

> An explicit instantiation of a function template shall not use the `inline` or `constexpr` specifiers.

This wording should be revised to apply to member functions of class templates as well.

Proposed resolution (February, 2010):

Change 17.8.2 [temp.explicit] paragraph 1 as follows:

> ...An explicit instantiation of a function template **or member function of a class template** shall not use the `inline` or `constexpr` specifiers.

---

995. Incorrect example for *using-declaration* and explicit instantiation

Section: 17.8.2 [temp.explicit] **Status:** CD2 **Submitter:** Doug Gregor **Date:** 27 Oct, 2009

[Voted into WP at March, 2010 meeting.]

17.8.2 [temp.explicit] paragraph 5 has an example that reads, in significant part,

```
namespace N {
  template<class T> class Y {
    void mf() { }
  };
}

using N::Y;
template class Y<int>; // OK: explicit instantiation in namespace N
```

In fact, paragraph 2 requires that an explicit instantiation with an unqualified name must appear in the same namespace in which the template was declared, so the example is ill-formed.

Proposed resolution (February, 2010):

Change the example in 17.8.2 [temp.explicit] paragraph 5 as follows:

```
namespace N {
  template<class T> class Y { void mf() { } };
}

template class Y<int>;            // error: class template Y not visible
                                  // in the global namespace

using N::Y;
~~template class Y<int>;~~            ~~// OK: explicit instantiation in namespace N~~
template class Y<int>;                // error: explicit instantiation outside of the
                                      // namespace of the template

template class N::Y<char*>;       // OK: explicit instantiation in namespace N
template void N::Y<double>::mf(); // OK: explicit instantiation
                                  // in namespace N
```

---

730. Explicit specializations of members of non-template classes

Section: 17.8.3 [temp.expl.spec] **Status:** CD2 **Submitter:** Bronek Kozicki **Date:** 3 October, 2008

N2800 comment DE 14

[Voted into WP at October, 2009 meeting.]

The list of entities that can be explicitly specialized in 17.8.3 [temp.expl.spec] paragraph 1 includes member templates of class templates but not member templates of non-template classes. This omission could lead to the conclusion that such member templates cannot be explicitly specialized. (Note, however, that paragraph 3 refers to "an explicit specialization for a member template of [a] class or class template.")

Proposed resolution (July, 2009):

Change 17.8.3 [temp.expl.spec] paragraph 1 as follows:

An explicit specialization of any of the following:

- ...
- member class template of a **class or** class template
- non-deleted member function template of a **class or** class template

can be declared...

---

884. Defining an explicitly-specialized static data member

Section: 17.8.3 [temp.expl.spec] **Status:** CD2 **Submitter:** Daniel Krügler **Date:** 29 April, 2009

[Voted into WP at October, 2009 meeting.]

17.8.3 [temp.expl.spec] paragraphs 15-16 contain the following note:

[*Note:* there is no syntax for the definition of a static data member of a template that requires default initialization.

```
template<> X Q<int>::x;
```

This is a declaration regardless of whether `X` can be default initialized (11.6 [dcl.init]). —*end note*]

While this note is still accurate, the C++0x list initialization syntax provides a way around the restriction, which could be useful if the class is not copyable or movable but has a default constructor. Perhaps the note should be updated to mention that possibility?

Proposed resolution (July, 2009):

Change 17.8.3 [temp.expl.spec] paragraphs 15-16 as follows:

An explicit specialization of a static data member of a template is a definition if the declaration includes an initializer; otherwise, it is a declaration. [*Note:* ~~there is no syntax for the~~ **The** definition of a static data member of a template that requires default initialization~~.~~ **must use a *braced-init-list:***

```
template<> X Q<int>::x;       // declaration
template<> X Q<int>::x ();    // error: declares a function
template<> X Q<int>::x {};    // definition
```

~~This is a declaration regardless of whether `X` can be default initialized (11.6 [dcl.init]).~~ —*end note*]

---

923. Inline explicit specializations

Section: 17.8.3 [temp.expl.spec] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 19 June, 2009

[Voted into WP at March, 2010 meeting.]

According to 17.8.3 [temp.expl.spec] paragraph 14,

An explicit specialization of a function template is inline only if it is explicitly declared to be...

This could be read to require that the `inline` keyword must appear in the declaration. However, 11.4 [dcl.fct.def] paragraph 10 says that a deleted function is implicitly inline, so it should be made clear that defining an explicit specialization as deleted makes it inline.

Proposed resolution (November, 2009):

Change 17.8.3 [temp.expl.spec] paragraph 14 as follows:

An explicit specialization of a function template is inline only if it is ~~explicitly~~ declared ~~to be~~ **with the** `inline` **specifier or defined as deleted**, and independently of whether its function template is **inline**. [*Example:*...

---

657. Abstract class parameter in synthesized declaration

Section: 17.9.2 [temp.deduct] **Status:** CD2 **Submitter:** Mike Miller **Date:** 31 October 2007

[Voted into WP at October, 2009 meeting.]

A customer of ours recently brought the following example to our attention. There's some question as to whether the Standard adequately addresses this example, and if it does, whether the outcome is what we'd like to see. Here's the example:

```
struct Abs {
  virtual void x() = 0;
};

struct Der: public Abs {
  virtual void x();
};

struct Cnvt {
  template <typename F> Cnvt(F);
};

void foo(Cnvt a);
void foo(Abs &a);

void f() {
  Der d;
  Abs *a = &d;
  foo(*a);        // #1
  return 0;
}
```

The question is how to perform overload resolution for the call at #1. To do that, we need to determine whether `foo(Cnvt)` is a viable function. That entails deciding whether there is an implicit conversion sequence that converts `Abs` (the type of `*a` in the call) to `Cnvt` (16.3.2 [over.match.viable] paragraph 3), and that involves a recursive invocation of overload resolution.

The initialization of the parameter of `foo(Cnvt)` is a case of copy-initialization of a class by user-defined conversion, so the candidate functions are the converting constructors of `Cnvt` (16.3.1.4 [over.match.copy] paragraph 1), of which there are two: the implicitly-declared copy constructor and the constructor template.

According to 17.8.1 [temp.inst] paragraph 8,

> If a function template or a member function template specialization is used in a way that involves overload resolution, a declaration of the specialization is implicitly instantiated (17.9.3 [temp.over]).

Template argument deduction results in "synthesizing" (17.9.3 [temp.over] paragraph 1) (or "instantiating," 17.8.1 [temp.inst] paragraph 8) the declaration

```
Cnvt::Cnvt(Abs)
```

Because `Abs` is an abstract class, this declaration violates the restriction of 13.4 [class.abstract] paragraph 3 ("An abstract class shall not be used as a parameter type..."), and because a parameter of an abstract class type does not cause a deduction failure (it's not in the bulleted list in 17.9.2 [temp.deduct] paragraph 2), the program is ill-formed. This error is reported by both EDG and Microsoft compilers, but not by g++.

It seems unfortunate that the program would be rendered ill-formed by a semantic violation in a declaration synthesized solely for the purpose of overload resolution analysis; `foo(Cnvt)` would not be selected by overload resolution, so `Cnvt::Cnvt(Abs)` would not be instantiated.

There's at least some indication that a parameter with an abstract class type should be a deduction failure; an array element of abstract class type is a deduction failure, so one might expect that a parameter would be, also.

(See also issue 339; this question might be addressed as part of the direction described in the notes from the July, 2007 meeting.)

Notes from the June, 2008 meeting:

Paper N2634, adopted at the June, 2008 meeting, replaces the normative list of specific errors accepted as deduction failures by a general statement covering all "invalid types and expressions in the immediate context of the function type and its template parameter types," so the code is now well-formed. However, the previous list is now a note, and the note should be updated to mention this case.

Proposed resolution (August, 2008):

Add a new bullet following the last bullet of the note in 17.9.2 [temp.deduct] paragraph 8 as follows:

- Attempting to create a function type in which a parameter type or the return type is an abstract class type (13.4 [class.abstract]).

---

847. Error in rvalue reference deduction example

Section: 17.9.2.1 [temp.deduct.call] **Status:** CD2 **Submitter:** Steve Adamczyk **Date:** 27 March, 2009

[Voted into WP at March, 2010 meeting.]

The adoption of paper N2844 made it ill-formed to attempt to bind an rvalue reference to an lvalue. However, the example in 17.9.2.1 [temp.deduct.call] paragraph 3 still reflects the previous specification:

```
template <typename T> int f(T&&);
int i;
int j = f(i);        // calls f<int&>(i)
template <typename T> int g(const T&&);
int k;
int n = g(k);        // calls g<int>(k)
```

The last line of that example is now ill-formed, attempting to bind the `const int&&` parameter of `g` to the lvalue `k`.

Proposed resolution (July, 2009):

Replace the example in 17.9.2.1 [temp.deduct.call] paragraph 3 with:

```
template<typename T> int f(T&&);
template<typename T> int g(const T&&);
int i;
int n1 = f(i);    // calls f<int&>(int&)
int n2 = f(0);    // calls f<int>(int&&)
int n3 = g(i);    // error: would call g<int>(const int&&), which would
                  // bind an rvalue reference to an lvalue
```

(See also issue 858.)

---

876. Type references in rvalue reference deduction specification

Section: 17.9.2.1 [temp.deduct.call] **Status:** CD2 **Submitter:** Steve Adamczyk **Date:** 20 April, 2009

[Voted into WP at October, 2009 meeting.]

17.9.2.1 [temp.deduct.call] paragraph 3 says,

> If `P` is of the form `T&&`, where `T` is a template parameter, and the argument is an lvalue, the type `A&` is used in place of `A` for type deduction.

The type references in that sentence are inconsistent with the normal usage in the Standard; they should instead refer to "an rvalue reference to a cv-unqualified template parameter" and "lvalue reference to `A`."

Proposed resolution (July, 2009):

Change 17.9.2.1 [temp.deduct.call] paragraph 3 as follows:

> If `P` is a cv-qualified type, the top level cv-qualifiers of `P`'s type are ignored for type deduction. If `P` is a reference type, the type referred to by `P` is used for type deduction. If `P` is ~~of the form `T&&`, where `T` is a template parameter,~~ **an rvalue reference to a cv-unqualified template parameter** and the argument is an lvalue, the type ~~`A&`~~ **"lvalue reference to `A`"** is used in place of `A` for type deduction.

---

493. Type deduction from a `bool` context

Section: 17.9.2.3 [temp.deduct.conv] **Status:** CD2 **Submitter:** John Spicer **Date:** 17 Dec 2004

[Voted into WP at March, 2010 meeting.]

An expression used in an `if` statement is implicitly converted to type `bool` (9.4 [stmt.select]). According to the rules of template argument deduction for conversion functions given in 17.9.2.3 [temp.deduct.conv], the following example is ill-formed:

```
struct X {
  template<class T> operator const T&() const;
};
int main()
{
  if( X() ) {}
}
```

Following the logic in 17.9.2.3 [temp.deduct.conv], `A` is `bool` and `P` is `const T` (because cv-qualification is dropped from `P` before the reference is removed), and deduction fails.

It's not clear whether this is the intended outcome or not.

Notes from the April, 2005 meeting:

The CWG observed that there is nothing special about either `bool` or the context in the example above; instead, it will be a problem wherever a copy occurs, because cv-qualification is always dropped in a copy operation. This appears to be a case where the conversion deduction rules are not properly symmetrical with the rules for arguments. The example should be accepted.

Proposed resolution (February, 2010):

This issue is resolved by the resolution of issue 976.

---

913. Deduction rules for array- and function-type conversion functions

Section: 17.9.2.3 [temp.deduct.conv] **Status:** CD2 **Submitter:** Steve Clamage **Date:** 10 June, 2009

[Voted into WP at March, 2010 meeting.]

The rules for deducing function template arguments from a conversion function template include provisions in 17.9.2.3 [temp.deduct.conv] paragraph 2 for array and function return types, even though such types are prohibited and cannot occur in the *conversion-type-id* of a conversion function template. They should be removed.

Proposed resolution (February, 2010):

This issue is resolved by the resolution of issue 976. In particular, under that resolution, if a conversion function returns a reference to an array or function type, the reference will be dropped prior to the adjustments mentioned in this issue, so they are, in fact, needed.

---

976. Deduction for `const T&` conversion operators

Section: 17.9.2.3 [temp.deduct.conv] **Status:** CD2 **Submitter:** Jens Maurer **Date:** 1 October, 2009

[Voted into WP at March, 2010 meeting.]

Consider this program:

```
struct F {
   template<class T>
```

```
   operator const T&() { static T t; return t; }
};

int main() {
   F f;
   int i = f;   // ill-formed
}
```

It's ill-formed, because according to 17.9.2.3 [temp.deduct.conv], we try to match `const T` with `int`.

(The reference got removed from `P` because of paragraph 3, but the `const` isn't removed, because paragraph 2 bullet 3 comes before paragraph 3 and thus isn't applied any more.)

Changing the declaration of the conversion operator to

```
operator T&() { ... }
```

makes the program compile, which is counter-intuitive to me: I'm in an rvalue (read-only) context, and I can use a conversion to `T&`, but I can't use a conversion to `const T&`?

Proposed resolution (February, 2010):

Change 17.9.2.3 [temp.deduct.conv] paragraphs 1-3 as follows, inserting a new paragraph between the current paragraphs 1 and 2:

> Template argument deduction is done by comparing the return type of the conversion function template (call it `P`**; see 11.6 [dcl.init], 16.3.1.5 [over.match.conv], and 16.3.1.6 [over.match.ref] for the determination of that type**) with the type that is required as the result of the conversion (call it `A`) as described in 17.9.2.5 [temp.deduct.type].
>
> **If `P` is a reference type, the type referred to by `P` is used in place of `P` for type deduction and for any further references to or transformations of `P` in the remainder of this section.**
>
> If `A` is not a reference type:
>
> - If `P` is an array type, the pointer type produced by the array-to-pointer standard conversion (7.2 [conv.array]) is used in place of `P` for type deduction; otherwise,
> - If `P` is a function type, the pointer type produced by the function-to-pointer standard conversion (7.3 [conv.func]) is used in place of `P` for type deduction; otherwise,
> - If `P` is a cv-qualified type, the top level cv-qualifiers of `P`'s type are ignored for type deduction.
>
> If `A` is a cv-qualified type, the top level cv-qualifiers of `A`'s type are ignored for type deduction. If `A` is a reference type, the type referred to by `A` is used for type deduction. ~~If `P` is a reference type, the type referred to by `P` is used for type deduction.~~

(This resolution also resolves issues 493 and 913.)

[Drafting note: This change intentionally reverses the resolution of issue 322 (and applies it in a different form).]

---

499. Throwing an array of unknown size

Section: 18.1 [except.throw] **Status:** CD2 **Submitter:** Mike Miller **Date:** 19 Jan 2005

[Voted into the WP at the March, 2009 meeting.]

According to 18.1 [except.throw] paragraph 3,

> The type of the *throw-expression* shall not be an incomplete type, or a pointer to an incomplete type other than (possibly cv-qualified) `void`.

This disallows cases like the following, because `str` has an incomplete type (an array of unknown size):

```
extern const char str[];
void f() {
    throw str;
}
```

The array-to-pointer conversion is applied to the operand of `throw`, so there's no problem creating the exception object, which is the reason for the restriction on incomplete types. I believe this case should be permitted.

Notes from the April, 2005 meeting:

The CWG agreed that the example should be permitted. Note that the reference to *throw-expression* in the cited text is incorrect; a *throw-expression* includes the `throw` keyword and is always of type `void`. This wording problem is addressed in the proposed resolution for issue 475.

Proposed resolution (October, 2006)

Change 18.1 [except.throw] paragraph 3 as indicated:

> ...~~The type of the *throw-expression* shall not~~ **If the type of the exception object would** be an incomplete type, or a pointer to an incomplete type other than (possibly cv-qualified) `void` **the program is ill-formed**...

---

828. Destruction of exception objects

Section: 18.1 [except.throw] **Status:** CD2 **Submitter:** UK **Date:** 3 March, 2009

N2800 comment UK 130

[Voted into WP at March, 2010 meeting.]

18.1 [except.throw] paragraph 4 says,

> When the last remaining active handler for the exception exits by any means other than `throw;` the temporary object is destroyed and the implementation may deallocate the memory for the temporary object...

With `std::current_exception()` (21.8.6 [propagation] paragraph 7), it might be possible to refer to the exception object after its last handler exits (if the exception object is not copied). The text needs to be updated to allow for that possibility.

Proposed resolution (September, 2009):

Change 18.1 [except.throw] paragraph 4 as follows:

> The memory for the ~~temporary copy of the exception being thrown~~ **exception object** is allocated in an unspecified way, except as noted in 6.7.4.1 [basic.stc.dynamic.allocation]. ~~The temporary persists as long as there is a handler being executed for that exception. In particular, if~~ **If** a handler exits by ~~executing a `throw;` statement, that passes control~~ **rethrowing, control is passed** to another handler for the same exception~~, so the temporary remains~~. **The exception object is destroyed after either** ~~When~~ the last remaining active handler for the exception exits by any means other than ~~`throw;`~~ **rethrowing, or the last object of type `std::exception_ptr` (21.8.6 [propagation]) that refers to the exception object is destroyed, whichever is later. In the former case, the destruction occurs when the handler exits, immediately after the destruction of the object declared in the *exception-declaration* in the handler, if any. In the latter case, the destruction occurs before the destructor of `std::exception_ptr` returns.** ~~the temporary object is destroyed and the~~ **The** implementation may **then** deallocate the memory for the ~~temporary~~ **exception** object; any such deallocation is done in an unspecified way. ~~The destruction occurs immediately after the destruction of the object declared in the *exception-declaration* in the handler.~~

---

830. Deprecating exception specifications

Section: 18.4 [except.spec] **Status:** CD2 **Submitter:** UK **Date:** 3 March, 2009

N2800 comment UK 136

[Voted into WP at March, 2010 meeting as paper N3051.]

Exception specifications have proven close to worthless in practice, while adding a measurable overhead to programs. The feature should be deprecated. The one exception to the rule is the empty throw specification which could serve a legitimate optimizing role if the requirement to call `std::unexpected` were relaxed in this case.

Notes from the July, 2009 meeting:

The consensus of the CWG was in favor of deprecating exception specifications. Further discussion, and with a wider constituency, is needed to determine a position on the status of `throw()`.

(See also issue 814.)

---

973. Function types in *exception-specification*s

Section: 18.4 [except.spec] **Status:** CD2 **Submitter:** Daniel Krügler **Date:** 12 October, 2009

[Voted into WP at March, 2010 meeting.]

There is no prohibition against specifying a function type in an *exception-specification*, and the normal conversion of a function type to a pointer-to-function type occurs in both *throw-expression*s (18.1 [except.throw] paragraph 3) and in *handler*s (18.3 [except.handle] paragraph 2), but that was apparently overlooked in the description of *exception-specification*s.

Proposed resolution (February, 2010):

Change 18.4 [except.spec] paragraphs 2-3 as follows:

> A type denoted in an *exception-specification* shall not denote an incomplete type. A type denoted in an *exception-specification* shall not denote a pointer or reference to an incomplete type, other than `void*`, `const void*`, `volatile void*`, or `const volatile void*`.

A type *cv* `T`, "array of `T`," or "function returning `T`" denoted in an *exception-specification* is adjusted to type `T`, "pointer to `T`," or "pointer to function returning `T`," respectively.

If any declaration of a function has an *exception-specification*, all declarations, including the definition and ~~an~~ **any** explicit specialization, of that function shall have an *exception-specification* with the same set of ~~*type-ids*~~ **adjusted types**. If any declaration of a pointer to function, reference to function, or pointer to member function has an *exception-specification*, all occurrences of that declaration shall have an *exception-specification* with the same set of ~~*type-ids*~~ **adjusted types**. In an explicit instantiation an *exception-specification* may be specified, but is not required. If an *exception-specification* is specified in an explicit instantiation directive, it shall have the same set of ~~*type-ids*~~ **adjusted types** as other declarations of that function. A diagnostic is required only if the sets of ~~*type-ids*~~ **adjusted types** are different within a single translation unit.

---

668. Throwing an exception from the destructor of a local static object

Section: 18.5.1 [except.terminate] **Status:** CD2 **Submitter:** Daniel Krügler **Date:** 16 December 2007

[Voted into the WP at the March, 2009 meeting.]

The destruction of local static objects occurs at the same time as that of non-local objects (6.6.3 [basic.start.dynamic] paragraph 1) and the execution of functions registered with `std::atexit` (paragraph 3). According to 18.5.1 [except.terminate] paragraph 1, `std::terminate` is called if a destructor for a non-local object or a function registered with `std::atexit` exits via an exception, but the Standard is silent about the result of throwing an exception from a destructor for a local static object. Presumably this is an oversight and the same rules should apply to destruction of local static objects.

Proposed resolution (September, 2008):

Change 18.5.1 [except.terminate] paragraph 1, fourth bullet as indicated, and add an additional bullet to follow it:

- when construction ~~or destruction~~ of a non-local object with static or thread storage duration exits using an exception (6.6.2 [basic.start.static]), or
- **when destruction of an object with static or thread storage duration exits using an exception (6.6.3 [basic.start.dynamic]), or**

---

601. Type of literals in preprocessing expressions

Section: 19.1 [cpp.cond] **Status:** CD2 **Submitter:** Daveed Vandevoorde **Date:** 23 October 2006

[Voted into WP at October, 2009 meeting.]

The description of preprocessing expressions in 19.1 [cpp.cond] paragraph 4 says,

> The resulting tokens comprise the controlling constant expression which is evaluated according to the rules of 5.19 using arithmetic that has at least the ranges specified in 21.3 [support.limits], except that all signed and unsigned integer types act as if they have the same representation as, respectively, `intmax_t` or `uintmax_t` (18.3.2).

However, this does not address the type implicitly assigned to integral literals. For example, in an implementation where `int` is 32 bits and `long long` is 64 bits, is a literal like `0xffffffff` signed or unsigned? WG14 adopted DR 265 to deal with this issue in the essentially-identical wording in C99; we should probably follow suit for C++.

Proposed Resolution (July, 2009):

Change 19.1 [cpp.cond] paragraph 4 as follows:

> ...and then each preprocessing token is converted into a token. The resulting tokens comprise the controlling constant expression which is evaluated according to the rules of 8.20 [expr.const] using arithmetic that has at least the ranges specified in 21.3 [support.limits]~~, except that~~**. For the purposes of this token conversion and evaluation** all signed and unsigned integer types act as if they have the same representation as, respectively, `intmax_t` or `uintmax_t` (N3035 .18.4.2 [stdinth]) **[*Footnote:* Thus on an implementation where `std::numeric_limits<int>::max()` is 0x7FFF and `std::numeric_limits<unsigned int>::max()` is 0xFFFF, the integer literal `0x8000` is signed and positive within a `#if` expression even though it is unsigned in translation phase 7 (5.2 [lex.phases]). —*end footnote*]**. This includes interpreting character literals...

---

618. Casts in preprocessor conditional expressions

Section: 19.1 [cpp.cond] **Status:** CD2 **Submitter:** Martin Sebor **Date:** 12 February 2007

[Voted into WP at October, 2009 meeting.]

19.1 [cpp.cond] paragraph 1 states,

> The expression that controls conditional inclusion shall be an integral constant expression except that: it shall not contain a cast...

The prohibition of casts is vacuous and misleading: as pointed out in the footnote in that paragraph,

> Because the controlling constant expression is evaluated during translation phase 4, all identifiers either are or are not macro names — there simply are no keywords, enumeration constants, and so on.

As a result, there can be no casts, which require either keywords or identifiers that resolve to types in order to be recognized as casts. The wording on casts should be removed and replaced by a note recognizing this implication.

Notes from the April, 2007 meeting:

The CWG agreed with this suggested resolution; however, the reference is in the "Preprocessing Directives" clause, which WG21 intends to keep in as close synchronization as possible with the corresponding wording in the C Standard. Any change here must therefore be done in consultation with WG14. Clark Nelson will fulfill this liaison function.

It was also noted that the imminent introduction of `constexpr` also has the potential for a similar kind of confusion, so the proposed resolution should address both casts and `constexpr`.

Proposed resolution (July, 2009):

Change 19.1 [cpp.cond] paragraph 1 as follows:

> The expression that controls conditional inclusion shall be an integral constant expression except that: ~~it shall not contain a cast;~~ identifiers (including those lexically identical to keywords)...

---

626. Preprocessor string literals

Section: 19.3.2 [cpp.stringize] **Status:** CD2 **Submitter:** Gennaro Prota **Date:** 12 September 2006

[Voted into WP at October, 2009 meeting.]

Clause 19 [cpp] refers in several places to "character string literals" without specifying whether they are narrow or wide strings. For instance, what kind of string does the `#` operator (19.3.2 [cpp.stringize]) produce?

19.4 [cpp.line] paragraph 1 says,

> The string literal of a `#line` directive, if present, shall be a character string literal.

Is "character string literal" intended to mean a narrow string literal? (Also, there is no *string-literal* mentioned in the grammatical descriptions of `#line`; paragraph 4 reads,

> `#` `line` *digit-sequence* ″ *s-char-sequence*$_{opt}$ ″ *new-line*

which is apparently intended to suggest a string literal but does not use the term.)

19.8 [cpp.predefined] should also specify what kind of character string literals are produced by the various string-valued predefined macros.

Notes from the July, 2007 meeting:

The CWG affirmed that all the string literals mentioned in Clause 19 [cpp] are intended to be narrow strings.

Proposed resolution (September, 2008)

1. Change the footnote in 19 [cpp] paragraph 1 as follows:

   > Thus, preprocessing directives are commonly called "lines." These "lines" have no other syntactic significance, as all white space is equivalent except in certain situations during preprocessing (see the `#` ~~character~~ string literal creation operator in 19.3.2 [cpp.stringize], for example).

2. Change 19.3.2 [cpp.stringize] paragraph 2 as follows:

   > If, in the replacement list, a parameter is immediately preceded by a `#` preprocessing token, both are replaced by a single ~~character~~ **ordinary** string literal **(5.13.5 [lex.string])** preprocessing token that contains the spelling of the preprocessing token sequence for the corresponding argument... Otherwise, the original spelling of each preprocessing token in the argument is retained in the ~~character~~ **ordinary** string literal, except for special handling for producing the spelling of string literals and character literals: a `\` character is inserted before each ″ and `\` character of a character literal or string literal (including the delimiting ″ characters). If the replacement that results is not a valid ~~character~~ **ordinary** string literal, the behavior is undefined. The ~~character~~ **ordinary** string literal corresponding to an empty argument is ″″. The order of evaluation of `#` and `##` operators is unspecified.

3. Change 19.3.5 [cpp.scope] paragraph 6 as follows:

   > To illustrate the rules for creating ~~character~~ **ordinary** string literals and concatenating tokens, the sequence... or, after concatenation of the ~~character~~ **ordinary** string literals...

4. Change 19.4 [cpp.line] paragraph 1 as follows:

   The string literal of a `#line` directive, if present, shall be ~~a character~~ **an ordinary** string literal.

5. Change 19.4 [cpp.line] paragraph 4 as follows:

   ...and changes the presumed name of the source file to be the contents of the ~~character~~ **ordinary** string literal.

6. Change 19.8 [cpp.predefined] paragraph 1 as follows:

   __DATE__

   The date of translation of the source file (~~a character~~ **an ordinary** string literal of the form...

   __FILE__

   The presumed name of the source file (~~a character~~ **an ordinary** string literal).

   ...

   __TIME__

   The time of translation of the source file (~~a character~~ **an ordinary** string literal of the form...

Notes from the September, 2008 meeting:

The proposed resolution will be discussed with the C Committee before proceeding, as it is expected that the next revision of the C Standard will also adopt new forms of string literals.

Additional notes (May, 2009):

At its most recent meeting, the C Committee decided to keep the existing term, "character string literal."

One possibility for maintaining compatible phraseology with the C Standard would be to replace the occurrences of "ordinary string literal" in 5.13.5 [lex.string] with "character string literal," instead of the extensive set of changes above.

Another possibility would be to leave the references in clause 19 [cpp] unchanged and just insert a prefatory comment near the beginning that every occurrence of "character string literal" refers to a *string-literal* with no prefix. (The use of "ordinary string literal" in the preceding edits is problematic in that the phrase includes raw string literals as well as unprefixed literals.)

Proposed resolution (July, 2009):

1. Change 19.3.2 [cpp.stringize] paragraph 2 as follows:

   A *character string literal* is a *string-literal* with no prefix. If, in the replacement list, a parameter is immediately preceded by a `#` preprocessing token...

2. Change the fifteenth bullet of Annex B [implimits] paragraph 2 as follows:

   - Characters in a ~~character~~ string literal ~~or wide-string literal~~ (after concatenation) [65 536].

---

831. Limit on recursively nested template instantiations

Section: B [implimits] **Status:** CD2 **Submitter:** DE **Date:** 3 March, 2009

N2800 comment DE 25

[Voted into WP at October, 2009 meeting.]

The limit of 17 recursively-nested template instantiations is too small for modern programming practices such as template metaprogramming. It is unclear, however, whether this is a useful metric; see this paper for an example that honors the limit but results in over 750 billion instantiations.

Notes from the July, 2009 meeting:

The consensus of the CWG was to increase the limit to 1024.

Proposed resolution (September, 2009):

Change B [implimits], the fourth bullet from the end, as follows:

- Recursively nested template instantiations [~~17~~ **1 024**].

Issues with "CD3" Status

1350. Incorrect exception specification for inherited constructors

Section: _N4527_.12.9 [class.inhctor] **Status:** CD3 **Submitter:** **Date:** 2011-08-16

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

According to _N4527_.12.9 [class.inhctor] paragraph 3, the exception specification for an inheriting constructor has the same exception specification as the inherited constructor. This ignores the exception specifications of default constructors for base classes and nonstatic data members and of functions called in *brace-or-equals-initializer*s of nonstatic data members.

Proposed resolution (August, 2011):

1. Delete the indicated bullet of _N4527_.12.9 [class.inhctor] paragraph 2:

   - ~~the *exception-specification* (18.4 [except.spec]),~~

2. Change _N4527_.12.9 [class.inhctor] paragraph 3 as follows:

   ...[*Note:* Default arguments are not inherited. **An *exception-specification* is implied as specified in 18.4 [except.spec].** *—end note*]

3. Change 18.4 [except.spec] paragraph 14 as follows:

   An **inheriting constructor (_N4527_.12.9 [class.inhctor]) and an** implicitly declared special member function (Clause 15 [special]) shall have an *exception-specification*. If `f` is **an inheriting constructor or** an implicitly declared default constructor, copy constructor, move constructor, destructor, copy assignment operator, or move assignment operator, its implicit *exception-specification* specifies the *type-id* `T` if and only if `T` is allowed by the *exception-specification* of a function directly invoked by `f`'s implicit definition; `f` shall allow all exceptions if any function it directly invokes allows all exceptions, and `f` shall allow no exceptions if every function it directly invokes allows no exceptions. **[*Note:* an instantiation of an inheriting constructor template has an implied *exception-specification* as if it were a non-template inheriting constructor. *—end note*]** [*Example*:...

1487. When are inheriting constructors declared?

Section: _N4527_.12.9 [class.inhctor] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2012-03-27

[Moved to DR at the April, 2013 meeting.]

According to _N4527_.12.9 [class.inhctor] paragraph 3,

> For each non-template constructor in the candidate set of inherited constructors other than a constructor having no parameters or a copy/move constructor having a single parameter, a constructor is implicitly declared with the same constructor characteristics unless there is a user-declared constructor with the same signature in the class where the *using-declaration* appears.

It is not clear whether that determination is intended to include constructors declared after the point of the *using-declaration* or not.

Proposed resolution (February, 2013):

1. Change 12.2 [class.mem] paragraph 2 as follows:

   A class is considered a completely-defined object type (6.9 [basic.types]) (or complete type) at the closing `}` of the *class-specifier*. Within the class *member-specification*, the class is regarded as complete within function bodies, default arguments, ***using-declaration*s introducing inheriting constructors (_N4527_.12.9 [class.inhctor]),** and *brace-or-equal-initializer*s for non-static data members (including such things in nested classes). Otherwise it is regarded as incomplete within its own class *member-specification*.

2. Change 15 [special] paragraph 1 as follows:

   ...See 15.1 [class.ctor], 15.4 [class.dtor] and 15.8 [class.copy]. *—end note*] **An implicitly-declared special member function is declared at the closing `}` of the *class-specifier*.** Programs shall not define implicitly-declared special member functions.

3. Change _N4527_.12.9 [class.inhctor] paragraph 3 as follows:

   For each non-template constructor in the candidate set of inherited constructors other than a constructor having no parameters or a copy/move constructor having a single parameter, a constructor is implicitly declared with the same constructor characteristics unless there is a user-declared constructor with the same signature in the **complete** class where the *using-declaration* appears. Similarly, for each constructor template in the candidate set of inherited constructors, a constructor template is implicitly declared with the same constructor characteristics unless there is an equivalent user-declared constructor template (17.6.6.1 [temp.over.link]) in the **complete** class where the *using-*

declaration appears. [*Note:* Default arguments are not inherited. An *exception-specification* is implied as specified in 18.4 [except.spec]. —*end note*]

Additional note (January, 2013):

A question has been raised as to whether it is necessary to prohibit inheriting constructors from base classes that are also enclosing classes when the derived class is defined outside the *member-specification* of the enclosing class. This issue has been returned to "review" status to allow discussion of this question.

Additional note (February, 2013):

It was observed that it is not permitted to derive from an incomplete class, which prevents the problem intended to be addressed by the prohibition of inheriting constructors from an enclosing class without disallowing such usage when the nested class is defined outside its enclosing class. That restriction has been removed from the proposed resolution.

1440. Acceptable *decltype-specifier*s used as *nested-name-specifier*s

Section: _N4567_.5.1.1 [expr.prim.general] **Status:** CD3 **Submitter:** Mike Miller **Date:** 2012-01-05

[Moved to DR at the October, 2012 meeting.]

The current wording of the Standard does not describe what happens when a *decltype-specifier* is used as a *nested-name-specifier* and the type denoted by the *decltype-specifier* is neither a class type nor an enumeration type. Such *nested-name-specifier*s should be prohibited, presumably somewhere around paragraphs 8-10 of _N4567_.5.1.1 [expr.prim.general]. (The corresponding prohibition for named types is handled as part of lookup in 6.4.3 [basic.lookup.qual] paragraph 1.)

Proposed resolution (February, 2012):

Add the following immediately after the grammar in _N4567_.5.1.1 [expr.prim.general] paragraph 8 and move the text following that point into a new paragraph:

> **The type denoted by a *decltype-specifier* in a *nested-name-specifier* shall be a class or enumeration type.**
>
> A *nested-name-specifier* that denotes a class...

---

616. Definition of "indeterminate value"

Section: 3 [intro.defs] **Status:** CD3 **Submitter:** Bjarne Stroustrup **Date:** 2 February 2007

[Moved to DR at the April, 2013 meeting.]

The C++ Standard uses the phrase "indeterminate value" without defining it. C99 defines it as "either an unspecified value or a trap representation." Should C++ follow suit?

In addition, 7.1 [conv.lval] paragraph 1 says that applying the lvalue-to-rvalue conversion to an "object [that] is uninitialized" results in undefined behavior; this should be rephrased in terms of an object with an indeterminate value.

Proposed resolution (October, 2012):

1. Change 7.1 [conv.lval] paragraphs 1 and 2 as follows (including changing the running text of paragraph 2 into bullets):

> A glvalue (6.10 [basic.lval]) of a non-function, non-array type `T` can be converted to a prvalue.[53] If `T` is an incomplete type, a program that necessitates this conversion is ill-formed. ~~If the object to which the glvalue refers is not an object of type `T` and is not an object of a type derived from `T`, or if the object is uninitialized, a program that necessitates this conversion has undefined behavior.~~ If `T` is a non-class type, the type of the prvalue is the cv-unqualified version of `T`. Otherwise, the type of the prvalue is `T`.[54]
>
> When an lvalue-to-rvalue conversion occurs in an unevaluated operand or a subexpression thereof (Clause 8 [expr]) the value contained in the referenced object is not accessed. **In all other cases, the result of the conversion is determined according to the following rules:**
>
> - **If `T` is (possibly cv-qualified) `std::nullptr_t`, the result is a null pointer constant (7.11 [conv.ptr]).**
> - Otherwise, if ~~the glvalue~~ `T` has a class type, the conversion copy-initializes a temporary of type `T` from the glvalue and the result of the conversion is a prvalue for the temporary.
> - **Otherwise, if the object to which the glvalue refers contains an invalid pointer value (6.7.4.2 [basic.stc.dynamic.deallocation], 6.7.4.3 [basic.stc.dynamic.safety]), the behavior is implementation-defined.**
> - **Otherwise, if `T` is a (possibly cv-qualified) unsigned character type (6.9.1 [basic.fundamental]), and the object to which the glvalue refers contains an indeterminate value (8.3.4 [expr.new], 11.6 [dcl.init], 15.6.2 [class.base.init]), and that object does not have automatic storage duration or the glvalue was the operand of a**

unary `&` operator or it was bound to a reference, the result is an unspecified value. [*Footnote:* The value may be different each time the lvalue-to-rvalue conversion is applied to the object. An `unsigned char` object with indeterminate value allocated to a register might trap. —*end footnote*]

- **Otherwise, if the object to which the glvalue refers contains an indeterminate value, the behavior is undefined.**
- ~~Otherwise, if the glvalue has (possibly cv-qualified) type `std::nullptr_t`, the prvalue result is a null pointer constant (7.11 [conv.ptr]).~~ Otherwise, the value contained in the object indicated by the glvalue is the prvalue result.

2. Change 8.2.5 [expr.ref] paragraph 4 second bullet as follows:

- If `E2` is a static data member...
- ...If `E1` is an lvalue, then `E1.E2` is an lvalue; ~~if `E1` is an xvalue, then~~ **otherwise** `E1.E2` is an xvalue~~; otherwise, it is a prvalue~~. Let the notation...
- If `E2` is a (possibly overloaded) member function...

3. Change 8.5 [expr.mptr.oper] paragraph 6 as follows:

> ...The result of a `.*` expression whose second operand is a pointer to a data member is ~~of the same value category (6.10 [basic.lval]) as its first operand~~ **an lvalue if the first operand is an lvalue and an xvalue otherwise**. The result of a `.*` expression whose second operand is a pointer to a member function...

This resolution also resolves issues 129, 240, 312, 623, and 1013.

(See also issue 1213.)

Additional note (August, 2012):

It was observed that the phrase in the fourth bullet of the change to 7.1 [conv.lval] paragraph 2 that reads "is not a local variable" should probably be changed to "does not have automatic storage duration," because objects with static storage duration are zero-initialized and thus cannot have an indeterminate value. The issue was returned to "review" status for discussion of this point.

1476. Definition of user-defined type

Section: 3 [intro.defs] **Status:** CD3 **Submitter:** Loïc Joly **Date:** 2012-03-08

[Moved to DR at the April, 2013 meeting.]

The Standard uses the phrase, "user-defined type," but it is not clear what it is intended to mean. For example, 20.5.4.2.1 [namespace.std] paragraph 1 says,

> A program may add a template specialization for any standard library template to namespace `std` only if the declaration depends on a user-defined type...

Are types defined in the Standard library "user-defined?"

10.1.7.2 [dcl.type.simple] paragraph 2 says,

> The `auto` specifier is a placeholder for a type to be deduced (10.1.7.4 [dcl.spec.auto]). The other *simple-type-specifier*s specify either a previously-declared user-defined type or one of the fundamental types (6.9.1 [basic.fundamental]).

implying that all non-fundamental types are "user-defined."

A definition is needed, as well as a survey of uses of the term to ensure consistency with the definition.

Proposed resolution (October, 2012):

1. Change 10.1.7.2 [dcl.type.simple] paragraph 2 as follows:

> The `auto` specifier is a placeholder for a type to be deduced (10.1.7.4 [dcl.spec.auto]). The other *simple-type-specifier*s specify either a previously-declared ~~user-defined~~ type**, a type determined from an expression,** or one of the fundamental types (6.9.1 [basic.fundamental]). Table 10 summarizes the valid combinations of *simple-type-specifier*s and the types they specify.

2. Change 7 [conv] paragraph 4 as follows:

> [*Note:* For ~~user-defined~~ **class** types, user-defined conversions are considered as well; see 15.3 [class.conv]. In general, an implicit conversion sequence (16.3.3.1 [over.best.ics]) consists of a standard conversion sequence followed by a user-defined conversion followed by another standard conversion sequence. —*end note*]

3. Change the example in 16.3.1.2 [over.match.oper] paragraph 1 as follows:

```
...
void f(void) {
  const char* p= "one" + "two";   // ill-formed because neither
                                  // operand has user-defined class or enumeration type
```

```
    int I = 1 + 1;              // Always evaluates to 2 even if
                                // ~~user-defined~~ class or enumeration types exist which
                                // would perform the operation.
}
```

1531. Definition of "access" (verb)

Section: 3 [intro.defs] **Status:** CD3 **Submitter:** Mike Miller **Date:** 2012-07-27

[Moved to DR at the April, 2013 meeting.]

The verb "access" is used in various places in the Standard (see 6.8 [basic.life] paragraphs 5 and 6 and 6.10 [basic.lval] paragraph 10) but is not defined. C99 defines it as

> <execution-time action> to read or modify the value of an object

(See also issue 1530.)

Proposed resolution (March, 2013):

1. Add the following to 3 [intro.defs]:

   access
   <execution-time action> to read or modify the value of an object

2. Change 4.6 [intro.execution] paragraph 12 as follows:

   ~~Accessing~~ **Reading** an object designated by a volatile glvalue (6.10 [basic.lval]), modifying an object, calling...

3. Change 4.7 [intro.multithread] paragraph 4 as follows:

   Two expression evaluations *conflict* if one of them modifies a memory location (4.4 [intro.memory]) and the other one ~~accesses~~ **reads** or modifies the same memory location.

4. Change 4.7 [intro.multithread] paragraph 24 as follows:

   The implementation may assume that any thread will eventually do one of the following:

   - ...
   - ~~access~~ **read** or modify a volatile object, or
   - ...

5. Change 8.18 [expr.ass] paragraph 8 as follows:

   If the value being stored in an object is ~~accessed from~~ **read via** another object that overlaps in any way the storage of the first object, then the overlap shall be exact and the two objects shall have the same type, otherwise the behavior is undefined. [*Note:*...

[Note: this wording was reviewed during the 2013-03-25 teleconference.]

---

129. Stability of uninitialized auto variables

Section: 4.6 [intro.execution] **Status:** CD3 **Submitter:** Nathan Myers **Date:** 26 June 1999

[Moved to DR at the April, 2013 meeting.]

Does the Standard require that an uninitialized auto variable have a stable (albeit indeterminate) value? That is, does the Standard require that the following function return `true`?

```
bool f() {
    unsigned char i;  // not initialized
    unsigned char j = i;
    unsigned char k = i;
    return j == k;    // true iff "i" is stable
}
```

6.9.1 [basic.fundamental] paragraph 1 requires that uninitialized `unsigned char` variables have a valid value, so the initializations of `j` and `k` are well-formed and required not to trap. The question here is whether the value of `i` is allowed to change between those initializations.

Mike Miller: 4.6 [intro.execution] paragraph 10 says,

> An instance of each object with automatic storage duration (6.7.3 [basic.stc.auto]) is associated with each entry into its block. Such an object exists and retains its last-stored value during the execution of the block and while the block is suspended...

I think that the most reasonable way to read this is that the only thing that is allowed to change the value of an automatic (non-volatile?) value is a "store" operation in the abstract machine. There are no "store" operations to `i` between the initializations of `j` and `k`, so it must

retain its original (indeterminate but valid) value, and the result of the program is well-defined.

The quibble, of course, is whether the wording "last-stored value" should be applied to a "never-stored" value. I think so, but others might differ.

Tom Plum: 10.1.7.1 [dcl.type.cv] paragraph 8 says,

> [*Note:* `volatile` is a hint to the implementation to avoid aggressive optimization involving the object because the value of the object might be changed by means undetectable by an implementation. See 4.6 [intro.execution] for detailed semantics. In general, the semantics of `volatile` are intended to be the same in C++ as they are in C.]

>From this I would infer that non-volatile means "shall not be changed by means undetectable by an implementation"; that the compiler is entitled to safely cache accesses to non-volatile objects if it can prove that no "detectable" means can modify them; and that therefore i *shall* maintain the same value during the example above.

Nathan Myers: This also has practical code-generation consequences. If the uninitialized auto variable lives in a register, and its value is *really* unspecified, then until it is initialized that register can be used as a temporary. Each time it's "looked at" the variable has the value that last washed up in that register. After it's initialized it's "live" and cannot be used as a temporary any more, and your register pressure goes up a notch. Fixing the uninit'd value would make it "live" the first time it is (or might be) looked at, instead.

Mike Ball: I agree with this. I also believe that it was certainly never my intent that an uninitialized variable be stable, and I would have strongly argued against such a provision. Nathan has well stated the case. And I am quite certain that it would be disastrous for optimizers. To ensure it, the frontend would have to generate an initializer, because optimizers track not only the lifetimes of variables, but the lifetimes of values assigned to those variables. This would put C++ at a significant performance disadvantage compared to other languages. Not even Java went this route. Guaranteeing defined behavior for a very special case of a generally undefined operation seems unnecessary.

Proposed resolution (February, 2012):

This issue is resolved by the resolution of issue 616.

---

912. Character literals and *universal-character-name*s

Section: 5.13.3 [lex.ccon] **Status:** CD3 **Submitter:** Alisdair Meredith **Date:** 7 June, 2009

[Moved to DR at the October, 2012 meeting.]

According to 5.13.3 [lex.ccon] paragraph 1,

> A character literal that does not begin with `u`, `U`, or `L` is an ordinary character literal, also referred to as a narrow-character literal. An ordinary character literal that contains a single *c-char* has type `char`, with value equal to the numerical value of the encoding of the c-char in the execution character set.

However, the definition of *c-char* includes as one possibility a *universal-character-name*. The value of a *universal-character-name* cannot, in general, be represented as a `char`, so this specification is impossible to satisfy.

(See also issue 411 for related questions.)

Additional note (February, 2012):

See the discussion in issue 1422 for a possible interpretation of the existing text.

Proposed resolution (February, 2012):

Change 5.13.3 [lex.ccon] paragraph 1 as follows:

> ...A character literal that does not begin with `u`, `U`, or `L` is an ordinary character literal, also referred to as a narrow-character literal. An ordinary character literal that contains a single *c-char* **representable in the execution character set** has type `char`, with value equal to the numerical value of the encoding of the *c-char* in the execution character set. An ordinary character literal that contains more than one *c-char* is a *multicharacter literal*. A multicharacter literal**, or an ordinary character literal containing a single *c-char* not representable in the execution character set, is conditionally-supported,** has type `int`**,** and **has an** implementation-defined value.

This resolution also resolves issue 1024.

1024. Limits on multicharacter literals

Section: 5.13.3 [lex.ccon] **Status:** CD3 **Submitter:** Alisdair Meredith **Date:** 2010-01-31

[Moved to DR at the October, 2012 meeting.]

There is no limit placed on the number of *c-char*s in a multicharacter literal or a wide-character literal containing multiple *c-char*s, either in 5.13.3 [lex.ccon] paragraphs 1-2 or in Annex B [implimits]. Presumably this means that an implementation must accept arbitrarily long

literals.

An alternative approach might be to state that these literals are conditionally supported with implementation-defined semantics, allowing an implementation to impose a documented limit that makes sense for the particular architecture.

Proposed resolution (February, 2012):

This issue is resolved by the resolution of issue 912.

---

712. Are integer constant operands of a *conditional-expression* "used?"

Section: 6.2 [basic.def.odr] **Status:** CD3 **Submitter:** Mike Miller **Date:** 9 September, 2008

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

In describing static data members initialized inside the class definition, 12.2.3.2 [class.static.data] paragraph 3 says,

> The member shall still be defined in a namespace scope if it is used in the program...

The definition of "used" is in 6.2 [basic.def.odr] paragraph 1:

> An object or non-overloaded function whose name appears as a potentially-evaluated expression is *used* unless it is an object that satisfies the requirements for appearing in a constant expression (8.20 [expr.const]) and the lvalue-to-rvalue conversion (7.1 [conv.lval]) is immediately applied.

Now consider the following example:

```
struct S {
  static const int a = 1;
  static const int b = 2;
};
int f(bool x) {
  return x ? S::a : S::b;
}
```

According to the current wording of the Standard, this example requires that `S::a` and `S::b` be defined in a namespace scope. The reason for this is that, according to 8.16 [expr.cond] paragraph 4, the result of this *conditional-expression* is an lvalue and the lvalue-to-rvalue conversion is applied to that, not directly to the object, so this fails the "immediately applied" requirement. This is surprising and unfortunate, since only the values and not the addresses of the static data members are used. (This problem also applies to the proposed resolution of issue 696.)

Proposed resolution (August, 2011):

Divide 6.2 [basic.def.odr] paragraph 2 into two paragraphs and change as follows:

An expression is *potentially evaluated* unless it is an unevaluated operand (Clause 5) or a subexpression thereof. **The *set of potential results* of an expression `e` is defined as:**

- **if `e` is an *id-expression* (_N4567_.5.1.1 [expr.prim.general]), the set whose sole member is `e`,**
- **if `e` is a class member access (8.2.5 [expr.ref]), the set of potential results of the object expression,**
- **if `e` is a pointer-to-member expression (8.5 [expr.mptr.oper]) whose second operand is a constant expression, the set of potential results of the object expression,**
- **if `e` has the form `(e1)`, the set of potential results of `e1`,**
- **if `e` is a glvalue conditional expression (8.16 [expr.cond]), the union of the sets of potential results of the second and third operands,**
- **if `e` is a comma expression (8.19 [expr.comma]), the set of potential results of the right operand,**
- **otherwise, the empty set.**

A variable `x` whose name appears as a potentially-evaluated expression `ex` is *odr-used* unless ~~it~~ `x` is an object that satisfies the requirements for appearing in a constant expression (8.20 [expr.const]) and `ex` **is an element of the set of potential results of an expression `e`, where either** the lvalue-to-rvalue conversion (7.1 [conv.lval]) is ~~immediately~~ applied **to `e`, or `e` is a discarded-value expression (Clause 8 [expr])**. `this` is odr-used...

[Drafting note: this wording requires `S::a` to be defined if it is used in an expression like `*&S::a`.]

1260. Incorrect use of term "overloaded" in description of odr-use

Section: 6.2 [basic.def.odr] **Status:** CD3 **Submitter:** Nikolay Ivchenkov **Date:** 2011-03-11

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

The current wording of 6.2 [basic.def.odr] paragraph 2 uses the term "overloaded" differently from its definition in 16 [over] paragraph 1. For example, names found by argument-dependent lookup are not "overloaded" if they are not declared in the same scope. The phrasing should be reconciled between the two sections.

Proposed resolution (August, 2011):

Change 6.2 [basic.def.odr] paragraph 2 as follows:

> ...A ~~non-overloaded~~ function whose name appears as a potentially-evaluated expression **is odr-used if it is the unique lookup result or it is the selected** ~~or a~~ member of a set of ~~candidate~~ **overloaded** functions **(6.4 [basic.lookup], 16.3 [over.match], 16.4 [over.over])**~~, if selected by overload resolution when referred to from a potentially-evaluated expression, is odr-used~~, unless it is a pure virtual function and its name is not explicitly qualified...

1362. Complete type required for implicit conversion to `T&`

Section: 6.2 [basic.def.odr] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2011-08-16

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

The requirement in 6.2 [basic.def.odr] paragraph 4 that a type `T` must be complete if an expression is implicitly converted to a pointer to `T` or reference to `T` inadvertently applies to user-defined conversions, although it was intended only to refer to built-in conversions.

Proposed resolution (August, 2011):

Change the indicated bullet of 6.2 [basic.def.odr] paragraph 4 as follows:

- an expression that is not a null pointer constant, and has type other than ***cv*** `void*`, is converted to the type pointer to `T` or reference to `T` using ~~an implicit~~ **a standard** conversion (Clause 7 [conv]), a `dynamic_cast` (8.2.7 [expr.dynamic.cast]) or a `static_cast` (8.2.9 [expr.static.cast]), or

---

1472. odr-use of reference variables

Section: 6.2 [basic.def.odr] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2012-03-01

[Moved to DR at the April, 2013 meeting.]

We have a special case in 6.2 [basic.def.odr] paragraph 2 that variables which satisfy the requirements for appearing in a constant expression are not odr-used if the lvalue-to-rvalue conversion is immediately applied. This special case only applies to objects, and thus does not apply to variables of reference type. This inconsistency seems strange, and there is implementation divergence:

```
int n;
void f() {
 constexpr int &r = n;
 [] { return r; }; // error: r is odr-used but not captured
}
```

This code is accepted by g++ but rejected by clang. Should `r` be odr-used here?

Proposed resolution (October, 2012):

Change 6.2 [basic.def.odr] paragraph 3 as follows:

> A variable `x` whose name appears as a potentially-evaluated expression `ex` is *odr-used* unless `x` ~~is an object that~~ satisfies the requirements for appearing in a constant expression (8.20 [expr.const]) and**, if** `x` **is an object,** `ex` is an element of the set of potential results of an expression `e`, where either the lvalue-to-rvalue conversion (7.1 [conv.lval]) is applied to `e`, or `e` is a discarded-value expression (Clause 8 [expr]). `this` is odr-used...

1511. `const volatile` variables and the one-definition rule

Section: 6.2 [basic.def.odr] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2012-06-18

[Moved to DR at the April, 2013 meeting.]

One of the criteria in 6.2 [basic.def.odr] paragraph 6 for when a entity is allowed to have multiple definitions is:

> in each definition of `D`, corresponding names, looked up according to 6.4 [basic.lookup], shall refer to an entity defined within the definition of `D`, or shall refer to the same entity, after overload resolution (16.3 [over.match]) and after matching of partial template specialization (17.9.3 [temp.over]), except that a name can refer to a `const` object with internal or no linkage if the object has the same literal type in all definitions of `D`, and the object is initialized with a constant expression (8.20 [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`; and

This wording is possibly not sufficiently clear for an example like:

```
const volatile int n = 0;
inline int get() { return n; }
```

Presumably this code could not appear in multiple translation units, because the requirement that `n` "has the same value in all definitions" cannot be satisfied (the value of a volatile variable can change "by means undetectable by the implementation," per 10.1.7.1 [dcl.type.cv] paragraph 7, so the value of `n` might be different in each translation unit). However, it might be good to make it explicit that "a `const` object" is not intended to apply to a volatile-qualified object.

Other points that were raised during the discussion of this issue were that it would probably be better to rephrase "the value (but not the address) of the object is used" in terms of the odr-use of the object, as well as questioning why a `const volatile` variable implicitly has internal linkage.

Proposed resolution (October, 2012):

1. Change 6.2 [basic.def.odr] paragraph 6 as follows:

   There can be more than one definition...

   - ...
   - in each definition of `D`, corresponding names, looked up according to 6.4 [basic.lookup], shall refer to an entity defined within the definition of `D`, or shall refer to the same entity, after overload resolution (16.3 [over.match]) and after matching of partial template specialization (17.9.3 [temp.over]), except that a name can refer to a **non-volatile** `const` object with internal or no linkage if the object has the same literal type in all definitions of `D`, and the object is initialized with a constant expression (8.20 [expr.const]), and ~~the value (but not the address) of~~ the object is ~~used~~ **not odr-used**, and the object has the same value in all definitions of `D`; and
   - ...

2. Change 6.5 [basic.link] paragraph 3 as follows:

   A name having namespace scope (6.3.6 [basic.scope.namespace]) has internal linkage if it is the name of

   - a variable, function or function template that is explicitly declared `static`; or,
   - a **non-volatile** variable that is explicitly declared `const` or `constexpr` and neither explicitly declared `extern` nor previously declared to have external linkage; or
   - a data member of an anonymous union.

1482. Point of declaration of enumeration

Section: 6.3.2 [basic.scope.pdecl] **Status:** CD3 **Submitter:** Daveed Vandevoorde **Date:** 2012-03-20

[Moved to DR at the April, 2013 meeting.]

According to 6.3.2 [basic.scope.pdecl] paragraph 2,

> The point of declaration for an enumeration is immediately after the identifier (if any) in either its *enum-specifier* (10.2 [dcl.enum]) or its first *opaque-enum-declaration* (10.2 [dcl.enum]), whichever comes first.

This would make the following example well-formed:

```
template<typename T> struct S { typedef char I; };
enum E: S<E>::I { e };
```

Presumably that would make `E` an incomplete enumeration type for the instantiation of `S<E>` (a concept not otherwise found in the Standard). However, some implementations reject this example, presumably making the point of declaration after the *enum-base*. We either need to make it ill-formed or describe incomplete enumeration types.

See also issue 977.

Proposed resolution (December, 2012):

1. Change 6.9 [basic.types] paragraph 5 as follows:

   A class that has been declared but not defined, **an enumeration type in certain contexts (10.2 [dcl.enum]),** or an array of unknown size or of incomplete element type, is an incompletely-defined object type.[43] Incompletely-defined object types...

2. Add the following as a new paragraph preceding 10.2 [dcl.enum] paragraph 6:

   An enumeration whose underlying type is fixed is an incomplete type from its point of declaration (6.3.2 [basic.scope.pdecl]) to immediately after its *enum-base* (if any), at which point it becomes a complete type. An enumeration whose underlying type is not fixed is an incomplete type from its point of declaration to immediately after the closing `}` of its *enum-specifier*, at which point it becomes a complete type.

For an enumeration whose underlying type is not fixed...

This resolution also resolves issue 977.

1352. Inconsistent class scope and completeness rules

Section: 6.3.7 [basic.scope.class] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2011-08-16

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

The rules regarding class scope and when the class is considered to be complete (normally implemented by deferred parsing of portions of class member declarations) are inconsistent and need to be clarified.

Proposed resolution (August, 2011):

1. Change 6.3.7 [basic.scope.class] paragraph 1 as follows:

    1. The potential scope of a name declared in a class consists not only of the declarative region following the name's point of declaration, but also of all function bodies, **default arguments, and** *brace-or-equal-initializer*s of non-static data members~~, and default arguments~~ in that class (including such things in nested classes).

2. Change 6.4.1 [basic.lookup.unqual] paragraph 7 as follows:

    A name used in the definition of a class X outside of a member function body**, default argument, *brace-or-equal-initializer* of a non-static data member,** or nested class definition[29] shall be declared in one of the following ways:...

3. Change 6.4.1 [basic.lookup.unqual] paragraph 8 as follows:

    ~~A~~ **For the members of a class** `X`**, a** name used **in a member function body, in a default argument, in the *brace-or-equal-initializer* of a non-static data member (12.2 [class.mem]), or** in the definition of a **class** member ~~function (12.2.1 [class.mfct]) of class `X`~~ **outside of the definition of** `X`**,** following the ~~function's~~ **member's** *declarator-id* [*Footnote:* That is, an unqualified name that occurs, for instance, in a type ~~or default argument~~ in the *parameter-declaration-clause* or in the ~~function body~~ ***exception-specification***. *—end footnote*]**,** ~~or in the *brace-or-equal-initializer* of a non-static data member (12.2 [class.mem]) of class `X`~~ shall be declared in one of the following ways:...

[Drafting note: 12.2 [class.mem] paragraph 2 requires no changes. 6.3.7 [basic.scope.class] paragraph 1 bullet 5 deals with out-of-class definitions, and bullet 2 ensures that the lookup results for argument types are the same for in-class and out-of-class declarations, so no change is required.]

1310. What is an "acceptable lookup result?"

Section: 6.4.3.1 [class.qual] **Status:** CD3 **Submitter:** Jason Merrill **Date:** 2011-05-06

[Moved to DR at the April, 2013 meeting.]

In 6.4.3.1 [class.qual] paragraph 2,

> In a lookup in which the constructor is an acceptable lookup result and the *nested-name-specifier* nominates a class `C`:
>
> - if the name specified after the *nested-name-specifier*, when looked up in `C`, is the injected-class-name of `C` (Clause 12 [class]), or
> - in a *using-declaration* (10.3.3 [namespace.udecl]) that is a *member-declaration*, if the name specified after the *nested-name-specifier* is the same as the *identifier* or the *simple-template-id*'s *template-name* in the last component of the *nested-name-specifier*,
>
> the name is instead considered to name the constructor of class `C`.

it is not clear what constitutes "an acceptable lookup result." For instance, is

```
struct S { } *sp = new S::S;
```

well-formed?

The intent of the wording was that `S::S` would refer to the constructor except in lookups that ignore the names of functions, e.g., in *elaborated-type-specifier*s and *nested-name-specifier*s. There doesn't seem to be a good reason to allow a *qualified-id* naming the injected-class-name. The alternative, i.e., only to find the constructor in a declarator, complicates parsing because the determination of whether the name is a type or a function would require lookahead.

Proposed resolution (August, 2012):

Change 6.4.3.1 [class.qual] paragraph 2 as follows:

> In a lookup in which ~~the constructor is an acceptable lookup result~~ **function names are not ignored [*Footnote:* Lookups in which function names are ignored include names appearing in a *nested-name-specifier*, an *elaborated-type-specifier*, or**

a *base-specifier*. —*end footnote*] and the *nested-name-specifier* nominates a class `C`:

- if the name specified after the *nested-name-specifier*, when looked up in `C`, is the injected-class-name of C (Clause 12 [class]), or
- in a *using-declaration* (10.3.3 [namespace.udecl]) that is a *member-declaration*, if the name specified after the *nested-name-specifier* is the same as the identifier or the *simple-template-id*'s *template-name* in the last component of the *nested-name-specifier*,

the name is instead considered to name the constructor of class `C`...

1415. Missing prohibition of block-scope definition of `extern` object

Section: 6.5 [basic.link] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2011-11-13

[Moved to DR at the October, 2012 meeting.]

There does not appear to be wording that prohibits a block-scope `extern` object declaration from being a definition.

Proposed resolution (February, 2012):

Add the following as a new paragraph following 11.6 [dcl.init] paragraph 4:

> [*Note:* Default arguments are more restricted; see 11.3.6 [dcl.fct.default].
>
> The order of initialization of variables with static storage duration is described in 6.6 [basic.start] and 9.7 [stmt.dcl]. —*end note*]
>
> **A declaration of a block-scope variable with external or internal linkage that has an *initializer* is ill-formed.**
>
> To *zero-initialize* an object...

1003. Acceptable definitions of `main`

Section: 6.6.1 [basic.start.main] **Status:** CD3 **Submitter:** Daniel Krügler **Date:** 2009-11-17

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

The specification of the forms of the definition of `main` that an impliementation is required to accept is clear in C99 that the parameter names and the exact syntactic form of the types can vary. Although it is reasonable to assume that a C++ implementation would accept a definition like

```
int main(int foo, char** bar) { /* ... */ }
```

instead of the canonical

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

it might be a good idea to clarify the intent using wording similar to C99's.

Proposed resolution (August, 2011):

Change 6.6.1 [basic.start.main] paragraph 2 as follows:

> ...All implementations shall allow both ~~of the following definitions of `main`:~~
>
> ~~`int main() { /* ... */ }`~~
>
> ~~and~~
>
> ~~`int main(int argc, char* argv[]) { /* ... */ }`~~
>
> - **function of `()` returning `int` and**
> - **function of `(int,` pointer to pointer to `char)` returning `int`**
>
> **as the type of `main` (11.3.5 [dcl.fct].** In the latter form**, for purposes of exposition, the first function parameter is called `argc` and the second function parameter is called `argv`, where `argc`** shall be the number of arguments...

1489. Is value-initialization of an array constant initialization?

Section: 6.6.2 [basic.start.static] **Status:** CD3 **Submitter:** Steve Adamczyk **Date:** 2012-03-29

[Moved to DR at the April, 2013 meeting.]

According to 6.6.2 [basic.start.static] paragraph 2,

> *Constant initialization* is performed:
>
> - ...
> - if an object with static or thread storage duration is not initialized by a constructor call and if every full-expression that appears in its initializer is a constant expression.

Presumably this would include a value-initialization (i.e., with no expressions) such as

```
int a[1000]{};
```

However, we have recently clarified the degenerate cases of other similar rules referencing "every," so it wouldn't hurt to be more explicit here.

Proposed resolution (October, 2012):

Change 6.6.2 [basic.start.static] paragraph 2 as follows:

> Constant initialization is performed:
>
> - ...
> - if an object with static or thread storage duration is not initialized by a constructor call and if **either the object is value-initialized or** every full-expression that appears in its initializer is a constant expression.

---

312. "use" of invalid pointer value not defined

Section: 6.7.4.2 [basic.stc.dynamic.deallocation] **Status:** CD3 **Submitter:** Martin von Loewis **Date:** 20 Sep 2001

[Moved to DR at the April, 2013 meeting.]

6.7.4.2 [basic.stc.dynamic.deallocation] paragraph 4 mentions that the effect of using an invalid pointer value is undefined. However, the standard never says what it means to 'use' a value.

There are a number of possible interpretations, but it appears that each of them leads to undesired conclusions:

1. A value is 'used' in a program if a variable holding this value appears in an expression that is evaluated. This interpretation would render the sequence

   ```
   int *x = new int(0);
   delete x;
   x = 0;
   ```

   into undefined behaviour. As this is a common idiom, this is clearly undesirable.
2. A value is 'used' if an expression evaluates to that value. This would render the sequence

   ```
   int *x = new int(0);
   delete x;
   x->~int();
   ```

   into undefined behaviour; according to 8.2.4 [expr.pseudo], the variable x is 'evaluated' as part of evaluating the pseudo destructor call. This, in turn, would mean that all containers (26 [containers]) of pointers show undefined behaviour, e.g. 26.3.10.4 [list.modifiers] requires to invoke the destructor as part of the `clear()` method of the container.

If any other meaning was intended for 'using an expression', that meaning should be stated explicitly.

(See also issue 623.)

Proposed resolution (February, 2012):

This issue is resolved by the resolution of issue 616.

---

623. Use of pointers to deallocated storage

Section: 6.7.4.2 [basic.stc.dynamic.deallocation] **Status:** CD3 **Submitter:** Herb Sutter **Date:** 27 February 2007

Any use of a pointer to deleted storage, even if the pointer is not dereferenced, produces undefined behavior (6.7.4.2 [basic.stc.dynamic.deallocation] paragraph 4). The reason for this restriction is that, on some historical architectures, deallocating an object might free a memory segment, resulting in a hardware exception if a pointer referring to that segment were loaded into a pointer register, and on those architectures use of a pointer register for moving and comparing pointers was the most efficient mechanism for these operations.

It is not clear whether current or foreseeable architectures still require such a draconian restriction or whether it is feasible to relax it only to forbid a smaller range of operations. Of particular concern is the use of atomic pointers, which might be used in race conditions involving deallocation, where the loser of the race might encounter this undefined behavior.

(See also issue 312.)

Rationale (April, 2007):

The current specification is clear and was well-motivated. Analysis of whether this restriction is still needed should be done via a paper and discussed in the Evolution Working Group rather than being handled by CWG as an issue/defect.

Additional note, February, 2014:

This issue was resolved by the resolution of issue 616, which made use of a pointer to deleted storage implementation-defined behavior.

1438. Non-dereference use of invalid pointers

Section: 6.7.4.3 [basic.stc.dynamic.safety] **Status:** CD3 **Submitter:** Anthony Williams **Date:** 2012-01-03

[Moved to DR at the October, 2012 meeting.]

The current Standard says that any use of an invalid pointer value produces undefined behavior (6.7.4.2 [basic.stc.dynamic.deallocation] paragraph 4). This includes not only dereferencing the pointer but even just fetching its value. The reason for this draconian restriction is that some architectures in the past used dedicated address registers for pointer loads and stores and they could fault if, for example, a segment number in a pointer was not currently mapped.

It is not clear whether such restrictions are necessary with architectures currently in use or reasonably foreseen. This should be investigated to see if the restriction can be loosened to apply only to dereferencing the pointer.

Proposed resolution (February, 2012):

Change 6.7.4.2 [basic.stc.dynamic.deallocation] paragraph 4 as follows:

> If the argument given to a deallocation function in the standard library is a pointer that is not the null pointer value (7.11 [conv.ptr]), the deallocation function shall deallocate the storage referenced by the pointer, rendering invalid all pointers referring to any part of the *deallocated storage*. ~~The effect of using an invalid pointer value (including passing it to a deallocation function) is undefined~~ **Indirection through an invalid pointer value and passing an invalid pointer value to a deallocation function have undefined behavior. Any other use of an invalid pointer value has implementation-defined behavior**. [*Footnote:* ~~On some~~ **Some** implementations~~, it~~ **might define that copying an invalid pointer value** causes a system-generated runtime fault. —*end footnote*]

---

597. Conversions applied to out-of-lifetime non-POD lvalues

Section: 6.8 [basic.life] **Status:** CD3 **Submitter:** Mike Miller **Date:** 27 September 2006

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

An lvalue referring to an out-of-lifetime non-POD class objects can be used in limited ways, subject to the restrictions in 6.8 [basic.life] paragraph 6:

> if the original object will be or was of a non-POD class type, the program has undefined behavior if:
>
> - the lvalue is used to access a non-static data member or call a non-static member function of the object, or
> - the lvalue is implicitly converted (7.11 [conv.ptr]) to a reference to a base class type, or
> - the lvalue is used as the operand of a `static_cast` (8.2.9 [expr.static.cast]) except when the conversion is ultimately to *cv* `char&` or *cv* `unsigned char&`), or
> - the lvalue is used as the operand of a `dynamic_cast` (8.2.7 [expr.dynamic.cast]) or as the operand of `typeid`.

There are at least a couple of questionable things in this list. First, there is no "implicit conversion to a reference to a base class," as assumed by the second bullet. Presumably this is intended to say that the lvalue is bound to a reference to a base class, and the cross-reference should be to 11.6.3 [dcl.init.ref], not to 7.11 [conv.ptr] (which deals with pointer conversions). However, even given that adjustment, it is not clear why it is forbidden to bind a reference to a non-virtual base class of an out-of-lifetime object, as that is just an address offset calculation. (Binding to a virtual base, of course, would require access to the value of the object and thus cannot be done outside the object's lifetime.)

The third bullet also appears questionable. It's not clear why `static_cast` is discussed at all here, as the only permissible `static_cast` conversions involving reference types and non-POD classes are to references to base or derived classes and to the same type, modulo cv-qualification; if implicit "conversion" to a base class reference is forbidden in the second bullet, why would an explicit conversion be

permitted in the third? Was this intended to refer to `reinterpret_cast`? Also, is there a reason to allow char types but disallow array-of-char types (which are more likely to be useful than a single char)?

Proposed resolution (March, 2008):

1. Change 6.8 [basic.life] paragraph 5 as follows:

   ...If the object will be or was of a non-trivial class type, the program has undefined behavior if:

   - the pointer is used to access a non-static data member or call a non-static member function of the object, or
   - the pointer is implicitly converted (7.11 [conv.ptr]) to a pointer to a **virtual** base class ~~type~~, or
   - the pointer is used as the operand of a `static_cast` (8.2.9 [expr.static.cast]) ~~(~~except when the conversion is to ~~`void*`, or to `void*` and subsequently to `char*`, or `unsigned char*`~~). **pointer to *cv* `void`, or to pointer to *cv* `void` and subsequently to pointer to *cv* `char` or pointer to *cv* `unsigned char`, or**
   - the pointer is used as the operand of a `dynamic_cast` (8.2.7 [expr.dynamic.cast])...

2. Change 6.8 [basic.life] paragraph 6 as follows:

   ...if the original object will be or was of a non-trivial class type, the program has undefined behavior if:

   - the lvalue is used to access a non-static data member or call a non-static member function of the object, or
   - the lvalue is ~~implicitly converted (7.11 [conv.ptr])~~ **bound** to a reference to a **virtual** base class ~~type~~ **(11.6.3 [dcl.init.ref])**, or
   - ~~the lvalue is used as the operand of a `static_cast` (8.2.9 [expr.static.cast]) except when the conversion is ultimately to *cv* `char&` or *cv* `unsigned char&`, or~~
   - the lvalue is used as the operand of a `dynamic_cast` (8.2.7 [expr.dynamic.cast]) or as the operand of `typeid`.

[Drafting notes: Paragraph 5 was changed to track the changes to paragraph 6. See also the resolution for issue 658.]

---

496. Is a volatile-qualified type really a POD?

Section: 6.9 [basic.types] **Status:** CD3 **Submitter:** John Maddock **Date:** 30 Dec 2004

[Moved to DR at the April, 2013 meeting.]

In 6.9 [basic.types] paragraph 10, the standard makes it quite clear that volatile qualified types are PODs:

> Arithmetic types (6.9.1 [basic.fundamental]), enumeration types, pointer types, and pointer to member types (6.9.2 [basic.compound]), and *cv-qualified* versions of these types (6.9.3 [basic.type.qualifier]) are collectively called *scalar types*. Scalar types, POD-struct types, POD-union types (clause 12 [class]), arrays of such types and *cv-qualified* versions of these types (6.9.3 [basic.type.qualifier]) are collectively called *POD types*.

However in 6.9 [basic.types] paragraph 3, the standard makes it clear that PODs can be copied "as if" they were a collection of bytes by `memcpy`:

> For any POD type `T`, if two pointers to `T` point to distinct `T` objects `obj1` and `obj2`, where neither `obj1` nor `obj2` is a base-class subobject, if the value of `obj1` is copied into `obj2`, using the `std::memcpy` library function, `obj2` shall subsequently hold the same value as `obj1`.

The problem with this is that a volatile qualified type may need to be copied in a specific way (by copying using only atomic operations on multithreaded platforms, for example) in order to avoid the "memory tearing" that may occur with a byte-by-byte copy.

I realise that the standard says very little about volatile qualified types, and nothing at all (yet) about multithreaded platforms, but nonetheless this is a real issue, for the following reason:

The forthcoming TR1 will define a series of traits that provide information about the properties of a type, including whether a type is a POD and/or has trivial construct/copy/assign operations. Libraries can use this information to optimise their code as appropriate, for example an array of type `T` might be copied with a `memcpy` rather than an element-by-element copy if `T` is a POD. This was one of the main motivations behind the type traits chapter of the TR1. However it's not clear how volatile types (or POD's which have a volatile type as a member) should be handled in these cases.

Notes from the April, 2005 meeting:

It is not clear whether the volatile qualifier actually guarantees atomicity in this way. Also, the work on the memory model for multithreading being done by the Evolution Working Group seems at this point likely to specify additional semantics for volatile data, and that work would need to be considered before resolving this issue.

(See also issue 1746.)

Proposed resolution, October, 2012:

1. Change 6.9 [basic.types] paragraph 9 as follows:

> ...Scalar types, trivially copyable class types (Clause 12 [class]), arrays of such types, and ~~cv-qualified~~ **non-volatile const-qualified** versions of these types (6.9.3 [basic.type.qualifier]) are collectively called *trivially copyable types*. Scalar types, trivial class types...

2. Change 10.1.7.1 [dcl.type.cv] paragraphs 6-7 as follows:

> **What constitutes an access to an object that has volatile-qualified type is implementation-defined.** If an attempt is made to refer to an object defined with a volatile-qualified type through the use of a glvalue with a non-volatile-qualified type, the program behavior is undefined.
>
> [*Note:* `volatile` is a hint to the implementation to avoid aggressive optimization involving the object because the value of the object might be changed by means undetectable by an implementation. **Furthermore, for some implementations,** `volatile` **might indicate that special hardware instructions are required to access the object.** See 4.6 [intro.execution] for detailed semantics. In general, the semantics of `volatile` are intended to be the same in C++ as they are in C. —*end note*]

3. Change 15.8 [class.copy] paragraph 12 as follows:

> A copy/move constructor for class `X` is trivial if it is not user-provided, its declared parameter type is the same as if it had been implicitly declared, and if
>
> - class `X` has no virtual functions (13.3 [class.virtual]) and no virtual base classes (13.1 [class.mi]), and
> - **class** `X` **has no non-static data members of volatile-qualified type, and**
> - ...

4. Change 15.8 [class.copy] paragraph 25 as follows:

> A copy/move assignment operator for class `X` is trivial if it is not user-provided, its declared parameter type is the same as if it had been implicitly declared, and if
>
> - class `X` has no virtual functions (13.3 [class.virtual]) and no virtual base classes (13.1 [class.mi]), and
> - **class** `X` **has no non-static data members of volatile-qualified type, and**
> - ...

Notes from the June, 2016 meeting:

The resolution of issue 2094 revert the changes above revert the changes of `volatile` qualification on trivial copyability.

1361. Requirement on *brace-or-equal-initializer*s of literal types

Section: 6.9 [basic.types] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2011-08-16

The requirement in 6.8 [basic.life] paragraph 10 that

- every constructor call and full-expression in the *brace-or-equal-initializer*s for non-static data members (if any) is a constant expression (8.20 [expr.const]),

is mostly redundant with the `constexpr` constructor requirements in 10.1.5 [dcl.constexpr] paragraph 4 (although 15.6.2 [class.base.init] does not establish a strict equivalence between *brace-or-equal-initializer*s and *mem-initializer*s).

Proposed resolution (April, 2013):

This issue is resolved by the changes in N3652, adopted at the April, 2013 (Bristol) meeting.

1405. `constexpr` and mutable members of literal types

Section: 6.9 [basic.types] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2011-10-21

[Moved to DR at the April, 2013 meeting.]

Currently, literal class types can have mutable members. It is not clear whether that poses any particular problems with constexpr objects and constant expressions, and if so, what should be done about it.

Proposed resolution (February, 2012):

Change 8.20 [expr.const] paragraph 2 as follows:

- ...

- an lvalue-to-rvalue conversion (7.1 [conv.lval]) unless it is applied to
  - a glvalue of integral or enumeration type that refers to a non-volatile const object with a preceding initialization, initialized with a constant expression, or
  - a glvalue of literal type that refers to a non-volatile object defined with `constexpr`, or that refers to a **non-mutable** sub-object of such an object, or
  - a glvalue of literal type that refers to a non-volatile temporary object whose lifetime has not ended, initialized with a constant expression;
- ...

1453. Volatile members in literal classes?

Section: 6.9 [basic.types] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2012-01-02

[Moved to DR at the October, 2012 meeting.]

Can a literal class have a volatile member? For example,

```
struct S {
  constexpr S() : n(0) { }
  volatile int n;
};

constexpr S s;   // causes volatile write to S::n
```

Proposed resolution (February, 2012):

Change 6.9 [basic.types] paragraph 10 as follows:

> A type is a *literal type* if it is:
>
> - ...
> - a class type (Clause 12 [class]) that has all the following properties:
>   - ...
>   - all of its non-static data members and base classes are of **non-volatile** literal types.

---

483. Normative requirements on integral ranges

Section: 6.9.1 [basic.fundamental] **Status:** CD3 **Submitter:** Steve Adamczyk **Date:** 21 Oct 2004

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

There is no normative requirement on the ranges of the integral types, although the footnote in 6.9.1 [basic.fundamental] paragraph 2 indicates the intent (for `int`, at least) that they match the values given in the `<climits>` header. Should there be an explicit requirement of some sort?

(See also paper N1693.)

Proposed resolution (August, 2011):

Change 6.9.1 [basic.fundamental] paragraph 3 as follows:

> ...collectively called the *extended integer types*. **The signed and unsigned integral types shall satisfy the constraints given in ISO C 5.2.4.2.1.**

1302. `noexcept` applied to expression of type `void`

Section: 6.9.1 [basic.fundamental] **Status:** CD3 **Submitter:** Nikolay Ivchenkov **Date:** 2011-04-22

[Moved to DR at the October, 2012 meeting.]

The list of acceptable uses of an expression of type `void` in 6.9.1 [basic.fundamental] paragraph 9 does not, but should, include an operand of the `noexcept` operator.

Proposed resolution (August, 2011):

Change 6.9.1 [basic.fundamental] paragraph 9 as follows:

> An expression of type `void` shall be used only as an expression statement (9.2 [stmt.expr]), as an operand of a comma expression (8.19 [expr.comma]), as a second or third operand of `?:` (8.16 [expr.cond]), as the operand of `typeid`, `noexcept`, or `decltype`, as the expression in a return statement (9.6.3 [stmt.return]) for a function with the return type `void`, or as the operand of an explicit conversion to type *cv* `void`.

1515. Modulo 2^n arithmetic for implicitly-unsigned types

Section: 6.9.1 [basic.fundamental] **Status:** CD3 **Submitter:** Sean Hunt **Date:** 2012-07-03

[Moved to DR at the April, 2013 meeting.]

According to 6.9.1 [basic.fundamental] paragraph 4,

> Unsigned integers, declared `unsigned`, shall obey the laws of arithmetic modulo 2^n where *n* is the number of bits in the value representation of that particular size of integer.

It is not clear whether this wording intentionally excludes types like `char16_t` and `char32_t` (and, possibly, types `char` and `wchar_t`, if those types are unsigned in a given implementation), since the `unsigned` keyword is not used in their declaration.

Proposed resolution (October, 2012):

Change 6.9.1 [basic.fundamental] paragraph 4 as follows:

> Unsigned integers~~, declared `unsigned`,~~ shall obey the laws of arithmetic modulo 2^n where *n* is the number of bits in the value representation of that particular size of integer.[46]

1539. Definition of "character type"

Section: 6.9.1 [basic.fundamental] **Status:** CD3 **Submitter:** Beman Dawes **Date:** 2012-08-15

[Moved to DR at the April, 2013 meeting.]

The term *character type* is used in the Standard without definition. It should be defined; however, the use of the term is divergent between the core and library clauses: in the former, it means narrow character types, while in the latter it includes `wchar_t`, `char16_t`, and `char32_t`, so care must be taken in ensuring that no inadvertent changes are implied.

Proposed resolution (October, 2012):

1. Change 6.7.4.3 [basic.stc.dynamic.safety] paragraph 1 as follows:

   > A *traceable pointer object* is
   >
   > - ...
   > - a sequence of elements in an array of **narrow** character type **(6.9.1 [basic.fundamental])**, where the size and alignment of the sequence match those of some object pointer type.

2. Change 6.9.1 [basic.fundamental] paragraph 1 as follows:

   > Objects declared as characters (char) shall be large enough to store any member of the implementation's basic character set. If a character from this set is stored in a character object, the integral value of that character object is equal to the value of the single character literal form of that character. It is implementation-defined whether a `char` object can hold negative values. Characters can be explicitly declared `unsigned` or `signed`. Plain `char`, `signed char`, and `unsigned char` are three distinct types**, collectively called *narrow character types***. A `char`, a `signed char`, and an `unsigned char` occupy the same amount of storage and have the same alignment requirements (6.11 [basic.align]); that is, they have the same object representation. For **narrow** character types, all bits of the object representation participate in the value representation. For unsigned **narrow** character types, all possible bit patterns of the value representation represent numbers. These requirements do not hold for other types. In any particular implementation, a plain `char` object can take on either the same values as a `signed char` or an `unsigned char`; which one is implementation-defined.

3. Change 6.11 [basic.align] paragraph 6 as follows:

   > The alignment requirement of a complete type can be queried using an `alignof` expression (8.3.6 [expr.alignof]). Furthermore, the ~~types `char`, `signed char`, and `unsigned char`~~ **narrow character types (6.9.1 [basic.fundamental])** shall have the weakest alignment requirement. [*Note:* This enables the **narrow** character types to be used as the underlying type for an aligned memory area (10.6.2 [dcl.align]). —*end note*]

4. Change 11.6.2 [dcl.init.string] paragraph 1 as follows:

~~A `char` array (whether plain `char`, `signed char`, or `unsigned char`)~~ **An array of narrow character type (6.9.1 [basic.fundamental])**, `char16_t` array, `char32_t` array, or `wchar_t` array can be initialized by a narrow ~~character~~ **string** literal, `char16_t` string literal, `char32_t` string literal, or wide string literal, respectively, or by an appropriately-typed string literal enclosed in braces **(5.13.5 [lex.string])**. Successive characters of the value of the string literal initialize the elements of the array. [*Example:*

1059. Cv-qualified array types (with rvalues)

Section: 6.9.3 [basic.type.qualifier] **Status:** CD3 **Submitter:** Nikolay Ivchenkov **Date:** 2010-03-20

[Moved to DR at the October, 2012 meeting.]

In spite of the resolution of issue 112, the exact relationship between cv-qualifiers and array types is not clear. There does not appear to be a definitive normative statement answering the question of whether an array with a const-qualified element type is itself const-qualified; the statement in 6.9.3 [basic.type.qualifier] paragraph 5,

> Cv-qualifiers applied to an array type attach to the underlying element type, so the notation "*cv* `T`," where `T` is an array type, refers to an array whose elements are so-qualified. Such array types can be said to be more (or less) cv-qualified than other types based on the cv-qualification of the underlying element types.

hints at an answer but is hardly decisive. For example, is the following example well-formed?

```
template <class T> struct is_const {
    static const bool value = false;
};
template <class T> struct is_const<const T> {
    static const bool value = true;
};

template <class T> void f(T &) {
    char checker[is_const<T>::value];
}

int const arr[1] = {};

int main() {
    f(arr);
}
```

Also, when 6.10 [basic.lval] paragraph 4 says,

> Class prvalues can have cv-qualified types; non-class prvalues always have cv-unqualified types.

does this apply to array rvalues, as it appears? That is, given

```
struct S {
    const int arr[10];
};
```

is the array rvalue `S().arr` an array of `int` or an array of `const int`?

(The more general question is, when the Standard refers to non-class types, should it be considered to include array types? Or perhaps only arrays of non-class types?)

Proposed resolution (December, 2011):

1. Change 6.9.3 [basic.type.qualifier] paragraph 5 as follows:

> ...Cv-qualifiers applied to an array type attach to the underlying element type, so the notation "*cv* `T`," where `T` is an array type, refers to an array whose elements are so-qualified. ~~Such array types can be said to be more (or less) cv-qualified than other types based on the cv-qualification of the underlying element types.~~ **An array type whose elements are cv-qualified is also considered to have the same cv-qualification as its elements. [*Example:***
>
> ```
> typedef char CA[5];
> typedef const char CC;
> CC arr1[5] = { 0 };
> const CA arr2 = { 0 };
> ```
>
> **The type of both `arr1` and `arr2` is "array of 5 `const char`," and the array type is considered to be `const`-qualified. —*end example*]**

2. Change 6.10 [basic.lval] paragraph 4 as follows:

> Class **and array** prvalues can have cv-qualified types; ~~non-class~~ **other** prvalues always have cv-unqualified types. Unless otherwise indicated...

1428. Dynamic const objects

Section: 6.9.3 [basic.type.qualifier] **Status:** CD3 **Submitter:** Daniel Krügler **Date:** 2011-12-08

[Moved to DR at the October, 2012 meeting.]

The definition of "const object" in 6.9.3 [basic.type.qualifier] paragraph 1 is:

> The presence of a `const` specifier in a *decl-specifier-seq* declares an object of *const-qualified object type*, such object is called a *const object*.

Because the type of an object created by a *new-expression* is given by a *type-id* or *new-type-id* rather than with a *decl-specifier-seq*, this definition gives the false impression that objects of dynamic storage duration cannot be const objects. The wording should be adjusted to make it clear that they can.

Proposed resolution (February, 2012):

1. Change 6.9.3 [basic.type.qualifier] paragraph 1 as follows:

> The term object type (4.5 [intro.object]) includes the cv-qualifiers specified **in the *decl-specifier-seq* (10.1 [dcl.spec]), *declarator* (Clause 11 [dcl.decl]), *type-id* (11.1 [dcl.name]), or *new-type-id* (8.3.4 [expr.new])** when the object is created. ~~The presence of a `const` specifier in a *decl-specifier-seq* declares an object of *const-qualified object type*, such object is called a *const object*. The presence of a `volatile` specifier in a *decl-specifier-seq* declares an object of *volatile-qualified object type*, such object is called a *volatile object*. The presence of both *cv-qualifiers* in a *decl-specifier-seq* declares an object of *const-volatile-qualified object type*, such object is called a *const volatile object*.~~
>
> - **A *const object* is an object of type `const T` or a non-mutable subobject of such an object.**
> - **A *volatile object* is an object of type `volatile T`, a subobject of such an object, or a mutable subobject of a const volatile object.**
> - **A *const volatile object* is an object of type `const volatile T`, a non-mutable subobject of such an object, a const subobject of a volatile object, or a non-mutable volatile subobject of a const object.**
>
> The cv-qualified or cv-unqualified versions of a type are distinct types; however, they shall have the same representation and alignment requirements (6.9 [basic.types]).[51]

2. Change 6.9.3 [basic.type.qualifier] paragraph 3 as follows:

> ~~Each non-static, non-mutable, non-reference data member of a const-qualified class object is const-qualified, each non-static, non-reference data member of a volatile-qualified class object is volatile-qualified and similarly for members of a const-volatile class.~~ See 11.3.5 [dcl.fct] and 12.2.2.1 [class.this] regarding function types...

---

240. Uninitialized values and undefined behavior

Section: 7.1 [conv.lval] **Status:** CD3 **Submitter:** Mike Miller **Date:** 8 Aug 2000

[Moved to DR at the April, 2013 meeting.]

7.1 [conv.lval] paragraph 1 says,

> If the object to which the lvalue refers is not an object of type `T` and is not an object of a type derived from `T`, or if the object is uninitialized, a program that necessitates this conversion has undefined behavior.

I think there are at least three related issues around this specification:

1. Presumably assigning a valid value to an uninitialized object allows it to participate in the lvalue-to-rvalue conversion without undefined behavior (otherwise the number of programs with defined behavior would be vanishingly small :-). However, the wording here just says "uninitialized" and doesn't mention assignment.
2. There's no exception made for `unsigned char` types. The wording in 6.9.1 [basic.fundamental] was carefully crafted to allow use of `unsigned char` to access uninitialized data so that `memcpy` and such could be written in C++ without undefined behavior, but this statement undermines that intent.
3. It's possible to get an uninitialized rvalue without invoking the lvalue-to-rvalue conversion. For instance:

```
struct A {
    int i;
    A() { } // no init of A::i
};
int j = A().i;  // uninitialized rvalue
```

   There doesn't appear to be anything in the current IS wording that says that this is undefined behavior. My guess is that we thought that in placing the restriction on use of uninitialized objects in the lvalue-to-rvalue conversion we were catching all possible cases, but we missed this one.

In light of the above, I think the discussion of uninitialized objects ought to be removed from 7.1 [conv.lval] paragraph 1. Instead, something like the following ought to be added to 6.9 [basic.types] paragraph 4 (which is where the concept of "value" is introduced):

> Any use of an indeterminate value (8.3.4 [expr.new], 11.6 [dcl.init], 15.6.2 [class.base.init]) of any type other than `char` or `unsigned char` results in undefined behavior.

John Max Skaller:

`A().i` had better be an lvalue; the rules are wrong. Accessing a member of a structure requires it be converted to an lvalue, the above calculation is 'as if':

```
struct A {
    int i;
    A *get() { return this; }
};
int j = (*A().get()).i;
```

and you can see the bracketed expression is an lvalue.

A consequence is:

```
int &j= A().i; // OK, even if the temporary evaporates
```

`j` now refers to a 'destroyed' value. Any use of `j` is an error. But the binding at the time is valid.

Proposed Resolution (November, 2006):

1. Add the indicated words to 6.9 [basic.types] paragraph 4:

> ... For trivial types, the value representation is a set of bits in the object representation that determines a value, which is one discrete element of an implementation-defined set of values. **Any use of an indeterminate value (8.3.4 [expr.new], 11.6 [dcl.init], 15.6.2 [class.base.init]) of a type other than** `unsigned char` **results in undefined behavior.**

2. Change 7.1 [conv.lval] paragraph 1 as follows:

> If the object to which the lvalue refers is not an object of type `T` and is not an object of a type derived from `T`, ~~or if the object is uninitialized,~~ a program that necessitates this conversion has undefined behavior.

Additional note (May, 2008):

The C committee is dealing with a similar issue in their DR338. According to this analysis, they plan to take almost the opposite approach to the one described above by augmenting the description of their version of the lvalue-to-rvalue conversion. The CWG did not consider that access to an unsigned char might still trap if it is allocated in a register and needs to reevaluate the proposed resolution in that light. See also issue 129.

Proposed resolution (February, 2012):

This issue is resolved by the resolution of issue 616.

1013. Uninitialized `std::nullptr_t` objects

Section: 7.1 [conv.lval] **Status:** CD3 **Submitter:** Miller **Date:** 2009-12-10

[Moved to DR at the April, 2013 meeting.]

According to 7.1 [conv.lval] paragraph 1, an lvalue-to-rvalue conversion on an uninitialized object produces undefined behavior. Since there is only one "value" of type `std::nullptr_t`, an lvalue-to-rvalue conversion on a `std::nullptr_t` glvalue does not need to fetch the value from storage. Is there any need for undefined behavior in this case?

Proposed resolution (February, 2012):

This issue is resolved by the resolution of issue 616.

1423. Convertibility of `nullptr` to `bool`

Section: 7.13 [conv.fctptr] **Status:** CD3 **Submitter:** Dave Abrahams **Date:** 2011-12-04

[Moved to DR at the October, 2012 meeting.]

The resolution of issue 654 (found in paper N2656) enabled conversion of rvalues of type `std::nullptr_t` to `bool`. It appears that the use cases for this conversion are primarily or exclusively the "contextually converted to `bool`" cases, with some possibility for inadvertent misuse in other contexts. Paper N2656 mentioned the idea of limiting the conversions to the direct initialization contexts; that possibility should be reexamined.

Proposed resolution (February, 2012):

Change 7.13 [conv.fctptr] paragraph 1 as follows:

> ...any other value is converted to `true`. ~~A~~ **For direct-initialization (11.6 [dcl.init]), a** prvalue of type `std::nullptr_t` can be converted to a prvalue of type `bool`; the resulting value is `false`.

1261. Explicit handling of cv-qualification with non-class prvalues

Section: 8 [expr] **Status:** CD3 **Submitter:** Nikolay Ivchenkov **Date:** 2011-03-12

[Moved to DR at the October, 2012 meeting.]

Proposed resolution (December, 2011):

1. Change 6.10 [basic.lval] paragraph 4 as follows (supersedes the corresponding change in the resolution of issue 1059):

   ~~Class prvalues can have cv-qualified types; non-class prvalues always have cv-unqualified types.~~ Unless otherwise indicated (8.2.2 [expr.call]), prvalues shall always have complete types or the `void` type; in addition to these types, glvalues can also have incomplete types. **[*Note:* class and array prvalues can have cv-qualified types; other prvalues always have cv-unqualified types. See Clause 8 [expr]. —*end note*]**

2. Add a new paragraph following 8 [expr] paragraph 5:

   If an expression initially has the type "reference to `T`" ...

   If a prvalue initially has the type "*cv* `T`," where `T` is a cv-unqualified non-class, non-array type, the type of the expression is adjusted to `T` prior to any further analysis.

3. Change 8.2.2 [expr.call] paragraph 3 as follows:

   If the *postfix-expression* designates a destructor (15.4 [class.dtor]), the type of the function call expression is `void`; otherwise, the type of the function call expression is the return type of the statically chosen function (i.e., ignoring the `virtual` keyword), even if the type of the function actually called is different. This **return** type shall be an object type, a reference type or ~~the type~~ ***cv*** `void`.

4. Change 8.2.3 [expr.type.conv] paragraph 2 as follows:

   ...[*Note:* if `T` is a non-class type that is cv-qualified, the *cv-qualifier*s are ~~ignored~~ **discarded** when determining the type of the resulting prvalue (~~6.10 [basic.lval]~~ **Clause 8 [expr]**). —*end note*]

5. Change 8.4 [expr.cast] paragraph 1 as follows:

   ...[*Note:* if `T` is a non-class type that is ~~*cv-qualified*~~ **cv-qualified**, the *cv-qualifier*s are ~~ignored~~ **discarded** when determining the type of the resulting prvalue; see ~~6.10 [basic.lval]~~ **Clause 8 [expr]**. —*end note*]

1383. Clarifying discarded-value expressions

Section: 8 [expr] **Status:** CD3 **Submitter:** Lawrence Crowl **Date:** 2011-08-30

[Moved to DR at the October, 2012 meeting.]

There are some points in the description discarded-value expressions that need clarification:

- Does this require the lvalue-to-rvalue conversion in the listed cases or only prohibit it in all other cases ("only if" vs "if and only if")?
- Does this apply only to built-in operations or to overloaded operators?
- Does this apply to non-POD types?

Suggested resolution:

In some contexts, an expression only appears for its side effects. Such an expression is called a *discarded-value expression*. The expression is evaluated and its value is discarded. The array-to-pointer (7.2 [conv.array]) and function-to-pointer (7.3 [conv.func]) standard conversions are not applied. The lvalue-to-rvalue conversion (7.1 [conv.lval]) is applied **if and** only if the expression is an lvalue of volatile-qualified type and it has one of the following forms:

- *id-expression* (_N4567_.5.1.1 [expr.prim.general]),
- ~~subscripting~~ **subscript operation** (8.2.1 [expr.sub]),
- class member access (8.2.5 [expr.ref]),
- indirection **operation** (8.3.1 [expr.unary.op]),
- pointer-to-member operation (8.5 [expr.mptr.oper]),
- conditional ~~expression~~ **operation** (8.16 [expr.cond]) where both the second and the third operands are one of ~~the above~~ **these forms**, or

- comma ~~expression~~ **operation** (8.19 [expr.comma]) where the right operand is one of ~~the above~~ **these forms**.

[*Note:* Expressions invoking user-defined operators are not the operations above. Discarded-value expressions apply to class types, which will be ill-formed if there is no volatile copy constructor with which to initialize the temporary. —*end note*]

Proposed resolution (February, 2012):

Change 8 [expr] paragraph 10 as follows:

...The lvalue-to-rvalue conversion (7.1 [conv.lval]) is applied **if and** only if the expression is an lvalue of volatile-qualified type and it ~~has~~ **is** one of the following ~~forms~~:

- (***expression***)**, where *expression* is one of these expressions,**
- *id-expression* (_N4567_.5.1.1 [expr.prim.general]),
- ...
- conditional expression (8.16 [expr.cond]) where both the second and the third operands are one of ~~the above~~ **these expressions**, or
- comma expression (8.19 [expr.comma]) where the right operand is one of ~~the above~~ **these expressions**.

[*Note:* Using an overloaded operator causes a function call; the above covers only operators with built-in meaning. If the lvalue is of class type, it must have a volatile copy constructor to initialize the temporary that is the result of the lvalue-to-rvalue conversion. —*end note*]

Additional note (February, 2012):

A problem was discovered that was not addressed by the proposed resolution that was reviewed at the February, 2012 meeting, so the issue has been moved back to "review" status with revised wording.

---

755. Generalized *lambda-capture*s

Section: 8.1.5 [expr.prim.lambda] **Status:** CD3 **Submitter:** John Freeman **Date:** 11 December, 2008

In the current specification of lambda expressions, a name appearing in a *lambda-capture* must refer to a local variable or reference with automatic storage duration (8.1.5 [expr.prim.lambda] paragraph 3). This restriction seems unnecessary and possibly confusing.

One possibility would be to extend the syntax of the *lambda-capture* to be something like

v = *expr*

with the meaning that the closure object would have a member named v initialized with the value *expr*. With this extension, the current syntax could be viewed as an abbreviation for

v = v

Rationale, March, 2009:

This idea was discussed and rejected by the EWG.

This issue was addressed by the adoption of N3648, adopted at the April, 2013 (Bristol) meeting.

---

974. Default arguments for lambdas

Section: 8.1.5 [expr.prim.lambda] **Status:** CD3 **Submitter:** Jason Merrill **Date:** 4 September, 2009

N3092 comment US 29

[Moved to DR status at the April, 2013 meeting.]

There does not appear to be any technical difficulty that would require the restriction in 8.1.5 [expr.prim.lambda] paragraph 5 against default arguments in *lambda-expression*s.

Proposed resolution (August, 2011):

Delete the following sentence from 8.1.5 [expr.prim.lambda] paragraph 5:

~~Default arguments (11.3.6 [dcl.fct.default]) shall not be specified in the *parameter-declaration-clause* of a *lambda-declarator*.~~

Additional note (February, 2012):

EWG requested that the adoption of this proposed resolution be postponed to allow them to discuss it. The issue has thus been returned to "review" status pending EWG action.

---

975. Restrictions on return type deduction for lambdas

Section: 8.1.5 [expr.prim.lambda] **Status:** CD3 **Submitter:** Jason Merrill **Date:** 4 September, 2009

N3092 comment US 30

[Moved to DR status at the April, 2013 meeting as part of paper N3638.]

There does not appear to be any technical difficulty that would require the current restriction that the return type of a lambda can be deduced only if the body of the lambda consists of a single return statement. In particular, multiple return statements could be permitted if they all return the same type.

Proposed resolution (August, 2011):

Change 8.1.5 [expr.prim.lambda] paragraph 4 as follows:

...If a *lambda-expression* does not include a *trailing-return-type*, it is as if the *trailing-return-type* denotes the following type:

- ~~if the compound-statement is of the form~~

  ~~{ *attribute-specifier-seq*$_{opt}$ `return` *expression* ;~~

  ~~the type of the returned expression after lvalue-to-rvalue conversion (7.1 [conv.lval]), array-to-pointer conversion (7.2 [conv.array]), and function-to-pointer conversion (7.3 [conv.func]);~~

- ~~otherwise, `void`;~~

- **if there are no `return` statements in the *compound-statement*, or all `return` statements return either an expression of type `void` or no *expression* or *braced-init-list*, the type `void`;**

- **otherwise, if all `return` statements return an expression and the types of the returned expressions after lvalue-to-rvalue conversion (7.1 [conv.lval]), array-to-pointer conversion (7.2 [conv.array]), and function-to-pointer conversion (7.3 [conv.func]) are the same, that common type;**

- **otherwise, the program is ill-formed.**

[*Example:*

```
auto x1 = [](int i){ return i; }; // OK: return type is int
auto x2 = []{ return { 1, 2 }; }; // error: the return type is void (a
                                  // braced-init-list is not an expression)
struct A { int fn1(); const int& fn2(); };
template <class T> void f () {
  [](T t, bool b){
    if (b)
      return t.fn1();
    else
      return t.fn2();
  };
}
template void f<A>();             // OK: lambda return type is int
```

—*end example*]

Additional note (February, 2012):

EWG requested that the adoption of this proposed resolution be postponed to allow them to discuss it. The issue has thus been returned to "review" status pending EWG action.

1048. `auto` **deduction and lambda return type deduction.**

Section: 8.1.5 [expr.prim.lambda] **Status:** CD3 **Submitter:** Jason Merrill **Date:** 2010-03-09

[Moved to DR at the April, 2013 meeting as part of paper N3638.]

`auto` and lambda return types use slightly different rules for determining the result type from an expression. `auto` uses the rules in 17.9.2.1 [temp.deduct.call], which explicitly drops top-level cv-qualification in all cases, while the lambda return type is based on the lvalue-to-rvalue conversion, which drops cv-qualification only for non-class types. As a result:

```
struct A { };

const A f();
```

```
auto a = f();               // decltype(a) is A
auto b = []{ return f(); }; // decltype(b()) is const A
```

This seems like an unnecessary inconsistency.

John Spicer:

The difference is intentional; `auto` is intended only to give a `const` type if you explicitly ask for it, while the lambda return type should generally be the type of the expression.

Daniel Krügler:

Another inconsistency: with `auto`, use of a *braced-init-list* can deduce a specialization of `std::initializer_list`; it would be helpful if the same could be done for a lambda return type.

Additional note, February, 2014:

EWG noted that g++ and clang differ in their treatment of this example and referred it back to CWG for resolution.

Proposed resolution, November, 2014:

This issue was actually resolved by paper N3638, adopted at the April, 2013 meeting. It is returned to "review" status to allow consideration of whether the resolution should be considered a change for C++14 or a retroactive change to C++11.

Notes from the November, 2014 meeting:

CWG agreed that the change embodied in paper N3638 should be considered to have been a DR against C++11.

1557. Language linkage of converted lambda function pointer

Section: 8.1.5 [expr.prim.lambda] **Status:** CD3 **Submitter:** Scott Meyers **Date:** 2012-09-19

[Moved to DR at the April, 2013 meeting.]

8.1.5 [expr.prim.lambda] paragraph 6 does not specify the language linkage of the function type of the closure type's conversion function.

Proposed resolution (October, 2012):

Change 8.1.5 [expr.prim.lambda] paragraph 6 as follows:

> The closure type for a *lambda-expression* with no *lambda-capture* has a public non-virtual non-explicit const conversion function to pointer to function **with C++ language linkage (10.5 [dcl.link]).** having the same parameter and return types as the closure type's function call operator. The value returned...

2162. Capturing `this` by reference

Section: 8.1.5 [expr.prim.lambda] **Status:** CD3 **Submitter:** Brian Bi **Date:** 2015-07-23

The current wording of 8.1.5 [expr.prim.lambda] paragraphs 15 and 16 does not, but presumably should, indicate that `this`, if implicitly captured, is always captured by copy and not by reference.

Proposed resolution (February, 2016):

Change 8.1.5 [expr.prim.lambda] paragraph 15 as follows, dividing the existing text into a bulleted list:

> An entity is *captured by copy* if
>
> - **it is** `this`**,**
> - it is implicitly captured and the *capture-default* is `=`**,** or
> - ~~if~~ it is explicitly captured with a capture that is not of the form `&` *identifier* or `&` *identifier initializer*.
>
> For each entity captured by copy...

Notes from the February, 2016 meeting:

Paper P0018 is intended to change this area, so this issue is being returned to "review" status to accommodate those changes.

1213. Array subscripting and xvalues

Section: 8.2.1 [expr.sub] **Status:** CD3 **Submitter:** Jason Merrill **Date:** 2010-10-24

[Moved to DR status at the April, 2013 meeting.]

Because the subscripting operation is defined as indirection through a pointer value, the result of a subscript operator applied to an xvalue array is an lvalue, not an xvalue. This could be surprising to some.

Proposed resolution (December, 2012):

Change 8.2.1 [expr.sub] paragraph 1 as follows:

> A postfix expression followed by an expression in square brackets is a postfix expression. One of the expressions shall have the type **"array of** `T` **" or** "pointer to T" and the other shall have unscoped enumeration or integral type. The result is ~~an lvalue~~ of type "`T`." The type "`T`" shall be a completely-defined object type.[62] The expression `E1[E2]` is identical (by definition) to `*((E1)+(E2))` [*Note:* see 8.3 [expr.unary] and 8.7 [expr.add] for details of `*` and `+` and 11.3.4 [dcl.array] for details of arrays. —*end note*]**, except that in the case of an array operand, the result is an lvalue if that operand is an lvalue and an xvalue otherwise.**

(See also issue 616.)

Additional note (January, 2013):

The preceding resolution differs from that discussed in the December, 2012 drafting review teleconference by the deletion of the words "an lvalue" in the second sentence. The issue has consequently been moved to "review" status instead of "tentatively ready."

1516. Definition of "virtual function call"

Section: 8.2.2 [expr.call] **Status:** CD3 **Submitter:** Nikolay Ivchenkov **Date:** 2012-07-05

[Moved to DR at the April, 2013 meeting.]

The terms "virtual function call" and "virtual call" are used in the Standard but are not defined.

Proposed resolution (August, 2012):

Change 8.2.2 [expr.call] paragraph 1 as follows:

> If the selected function is non-virtual, or if the *id-expression* in the class member access expression is a *qualified-id*, that function is called. Otherwise, its final overrider (13.3 [class.virtual]) in the dynamic type of the object expression is called**; such a call is referred to as a *virtual function call***.

1269. `dynamic_cast` of an xvalue operand

Section: 8.2.7 [expr.dynamic.cast] **Status:** CD3 **Submitter:** Michael Wong **Date:** 2011-03-21

[Moved to DR at the October, 2012 meeting.]

8.2.7 [expr.dynamic.cast] paragraph 2 allows an expression of any value category when the target type is an rvalue reference. However, paragraph 6 requires that the operand be an lvalue if the runtime check is to be applied. This requirement should presumably be relaxed to require only a glvalue when the target type is an rvalue reference.

Proposed resolution (August, 2011):

Change 8.2.7 [expr.dynamic.cast] paragraph 6 as follows:

> Otherwise, `v` shall be a pointer to or ~~an lvalue~~ **a glvalue** of a polymorphic type (13.3 [class.virtual]).

Additional note, January, 2012:

An objection has been raised to the proposed resolution on the basis that it unnecessarily weakens the distinction between rvalues and lvalues, making it easier to create dangling references. Its status has therefore been changed back to "review" to allow further discussion.

1416. Function cv-qualifiers and `typeid`

Section: 8.2.8 [expr.typeid] **Status:** CD3 **Submitter:** Daniel Krügler **Date:** 2011-11-17

[Moved to DR at the October, 2012 meeting.]

The requirement in 8.2.8 [expr.typeid] paragraph 5 that

The top-level cv-qualifiers of the glvalue expression or the *type-id* that is the operand of `typeid` are always ignored

could be misinterpreted as referring to cv-qualifiers in a function type, even though it is clear that a function type is never cv-qualified. A note emphasizing the fact that that is not the case would be helpful.

Proposed resolution (February, 2012):

Change 8.2.8 [expr.typeid] paragraph 5 as follows:

> ~~The top-level cv-qualifiers of the glvalue expression or the *type-id* that is the operand of `typeid` are always ignored.~~ **If the type of the expression or *type-id* is a cv-qualified type, the result of the `typeid` expression refers to a `std::type_info` object representing the cv-unqualified type.** [*Example:*...

1320. Converting scoped enumerations to `bool`

Section: 8.2.9 [expr.static.cast] **Status:** CD3 **Submitter:** Jonathan Wakely **Date:** 2011-05-18

[Moved to DR at the April, 2013 meeting.]

The specification of `static_cast` (8.2.9 [expr.static.cast]) does not describe conversion of a scoped enumeration value to `bool`. Presumably it should be handled as for unscoped enumerations, with a zero value becoming `false` and a non-zero value becoming `true`.

Proposed resolution (August, 2012):

Change 8.2.9 [expr.static.cast] paragraph 9 as follows:

> A value of a scoped enumeration type (10.2 [dcl.enum]) can be explicitly converted to an integral type. ~~The~~ **When that type is *cv* `bool`, the resulting value is `false` if the original value is zero and `true` for all other values. For the remaining integral types, the** value is unchanged if the original value can be represented by the specified type. Otherwise, the resulting value is unspecified.

1412. Problems in specifying pointer conversions

Section: 8.2.9 [expr.static.cast] **Status:** CD3 **Submitter:** Nikolay Ivchenkov **Date:** 2011-11-01

[Moved to DR at the April, 2013 meeting.]

The definedness of various pointer conversions (see 8.2.10 [expr.reinterpret.cast] paragraph 7, 12.2 [class.mem] paragraph 20, 8.2.9 [expr.static.cast] paragraph 13) relies on the properties of the types involved, such as whether they are standard-layout types and their intrinsic alignment. This creates contradictions and unnecessary unspecified behavior when the actual values of the pointer involved would actually permit the operations. Recasting the specification in terms of the memory model instead of the types of the objects provides a more coherent specification.

Proposed resolution (February, 2012):

1. Change 7.11 [conv.ptr] paragraph 2 as follows:

> A prvalue of type "pointer to *cv* `T`," where `T` is an object type, can be converted to a prvalue of type "pointer to *cv* `void`". The result of converting a ~~"pointer to *cv* `T`"~~ **non-null pointer value of a pointer to object type** to a "pointer to *cv* void" ~~points to the start of the storage location where the object of type `T` resides, as if the object is a most derived object (4.5 [intro.object]) of type `T` (that is, not a base-class subobject)~~ **represents the address of the same byte in memory as the original pointer value**. The null pointer value is converted to the null pointer value of the destination type.

2. Change 8.2.9 [expr.static.cast] paragraph 13 as follows:

> A prvalue of type "pointer to *cv1* `void`" can be converted to a prvalue of type "pointer to *cv2* `T`," where `T` is an object type and *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*. The null pointer value is converted to the null pointer value of the destination type. **If the original pointer value represents the address `A` of a byte in memory and `A` satisfies the alignment requirement of `T`, then the resulting pointer value represents the same address as the original pointer value, that is, `A`. The result of any other such pointer conversion is unspecified.** A value of type pointer to object converted to "pointer to *cv* `void`" and back, possibly with different cv-qualification, shall have its original value...

3. Change 8.2.10 [expr.reinterpret.cast] paragraph 7 as follows:

> An object pointer can be explicitly converted to an object pointer of a different type.[70] When a prvalue `v` of **object pointer** type ~~"pointer to `T1`"~~ is converted to the **object pointer** type "pointer to *cv* `T2`", the result is `static_cast<`*cv* `T2*>(static_cast<`*cv* `void*>(v))` ~~if both `T1` and `T2` are standard-layout types (6.9 [basic.types]) and the alignment requirements of `T2` are no stricter than those of `T1`, or if either type is `void`~~. Converting a prvalue of type "pointer to `T1`" to the type "pointer to `T2`" (where `T1` and `T2` are object types and where the alignment requirements of `T2` are no stricter than those of `T1`) and back to its original type yields the original pointer value. ~~The result of any other such pointer conversion is unspecified.~~

1447. `static_cast` **of bit-field lvalue to rvalue reference**

Section: 8.2.9 [expr.static.cast] **Status:** CD3 **Submitter:** Jason Merrill **Date:** 2012-01-16

[Moved to DR at the October, 2012 meeting.]

According to 8.2.9 [expr.static.cast] paragraph 3,

> A glvalue of type "*cv1* `T1`" can be cast to type "rvalue reference to *cv2* `T2`," if "*cv2* `T2`" is reference-compatible with "*cv1* `T1`" (11.6.3 [dcl.init.ref]). The result refers to the object or the specified base class subobject thereof.

This specification fails to allow for a bit-field lvalue operand, since the reference cannot refer to a bit-field. Presumably a temporary should be formed and the reference be bound to it.

Proposed resolution (February, 2012):

Change 8.2.9 [expr.static.cast] paragraphs 3-4 as follows:

> A glvalue of type "*cv1* `T1`" can be cast to type "rvalue reference to *cv2* `T2`" if "*cv2* `T2`" is reference-compatible with "*cv1* `T1`" (11.6.3 [dcl.init.ref]). ~~The~~ **If the glvalue is not a bit-field, the** result refers to the object or the specified base class subobject thereof**; otherwise, the lvalue-to-rvalue conversion (7.1 [conv.lval]) is applied to the bit-field and the resulting prvalue is used as the *expression* of the** `static_cast` **for the remainder of this section**. If `T2` is an inaccessible (Clause 14 [class.access]) or ambiguous (13.2 [class.member.lookup]) base class of `T1`, a program that necessitates such a cast is ill-formed.
>
> ~~Otherwise, an~~ **An** expression `e` can be explicitly converted...

1268. `reinterpret_cast` **of an xvalue operand**

Section: 8.2.10 [expr.reinterpret.cast] **Status:** CD3 **Submitter:** Michael Wong **Date:** 2011-03-21

[Moved to DR at the October, 2012 meeting.]

8.2.10 [expr.reinterpret.cast] paragraph 11, dealing with casting to reference types, only allows an lvalue operand. Presumably it should allow a glvalue operand when the target is an rvalue reference type.

Proposed resolution (August, 2011):

Change 8.2.10 [expr.reinterpret.cast] paragraph 11:

> ~~An lvalue~~ **A glvalue** expression of type `T1` can be cast to the type "reference to `T2`" if an expression of type "pointer to `T1`" can be explicitly converted to the type "pointer to `T2`" using a `reinterpret_cast`. **The result refers to the same object as the source glvalue, but with the specified type. [*Note:* That is, for lvalues,** a reference cast `reinterpret_cast<T&>(x)` has the same effect as the conversion `*reinterpret_cast<T*>(&x)` with the built-in & and * operators (and similarly for `reinterpret_cast<T&&>(x)`). **—** ***end note*****]** ~~The result refers to the same object as the source lvalue, but with a different type. The result is an lvalue for an lvalue reference type or an rvalue reference to function type and an xvalue for an rvalue reference to object type.~~ No temporary is created,...

Additional note, January, 2012:

An objection has been raised to the proposed resolution on the basis that it unnecessarily weakens the distinction between rvalues and lvalues, making it easier to create dangling references. Its status has therefore been changed back to "review" to allow further discussion.

---

342. Terminology: "indirection" versus "dereference"

Section: 8.3 [expr.unary] **Status:** CD3 **Submitter:** Jason Merrill **Date:** 7 Oct 2001

[Moved to DR at the October, 2012 meeting.]

Split off from issue 315.

Incidentally, another thing that ought to be cleaned up is the inconsistent use of "indirection" and "dereference". We should pick one.

Proposed resolution (December, 2006):

1. Change 8.3.1 [expr.unary.op] paragraph 1 as follows:

> The unary * operator ~~performs *indirection*~~ **dereferences a pointer value**: the expression to which it is applied shall be a pointer...

2. Change 11.3.4 [dcl.array] paragraph 8 as follows:

> The ~~results are added and indirection applied~~ **values are added and the result is dereferenced** to yield an array (of five integers), which in turn is converted to a pointer to the first of the integers.

3. Change 11.3.5 [dcl.fct] paragraph 9 as follows:

> The binding of `*fpi(int)` is `*(fpi(int))`, so the declaration suggests, and the same construction in an expression requires, the calling of a function `fpi`, and then ~~using indirection through~~ **dereferencing** the (pointer) result to yield an integer. In the declarator `(*pif)(const char*, const char*)`, the extra parentheses are necessary to indicate that ~~indirection through~~ **dereferencing** a pointer to a function yields a function, which is then called.

4. Change the index for * and "dereferencing" no longer to refer to "indirection."

[*Drafting note:* 29.7.9 [template.indirect.array] requires no change. Many more places in the current wording use "dereferencing" than "indirection."]

Notes from the August, 2011 meeting:

CWG prefers use of the term "indirection" instead of "dereferencing." This would be consistent with the usage in the C Standard and would avoid entanglement with the C++ concept of a reference type.

Proposed resolution (February, 2012):

1. Change 6.7.4.1 [basic.stc.dynamic.allocation] paragraph 2 as follows:

> ...The effect of ~~dereferencing~~ **indirecting through** a pointer returned as a request for zero size is undefined.

2. Change 6.7.4.3 [basic.stc.dynamic.safety] paragraph 2 as follows:

> - the value returned by a call to the C ++ standard library implementation of `::operator new(std:: size_t)`; [*Footnote:* This section does not impose restrictions on ~~dereferencing~~ **indirection through** pointers to memory not allocated by `::operator new`. This maintains the ability of many C++ implementations to use binary libraries and components written in other languages. In particular, this applies to C binaries, because ~~dereferencing~~ **indirection through** pointers to memory allocated by `std::malloc` is not restricted. —*end footnote*]
> - the result of taking the address of an object (or one of its subobjects) designated by an lvalue resulting from ~~dereferencing~~ **indirection through** a safely-derived pointer value;
> - ...

3. Change 6.8 [basic.life] paragraph 5 as follows:

> ...~~Such~~ **Indirection through such** a pointer ~~may be dereferenced~~ **is permitted** but the resulting lvalue may only be used in limited ways...

4. Change 7.12 [conv.mem] paragraph 2 as follows:

> ...Since the result has type "pointer to member of `D` of type *cv* `T`", ~~it can be dereferenced~~ **indirection through it** with a `D` object **is valid**. The result is the same as if **indirecting through** the pointer to member of `B` ~~were dereferenced~~ with the `B` subobject of `D`. The null member pointer value...

5. Change 8.2.9 [expr.static.cast] paragraph 12 as follows:

> ...[*Note:* although class `B` need not contain the original member, the dynamic type of the object ~~on~~ **with** which **indirection through** the pointer to member is ~~dereferenced~~ **performed** must contain the original member; see 8.5 [expr.mptr.oper]. —*end note*]

6. Change 8.3.1 [expr.unary.op] paragraph 1 as follows:

> ...[*Note:* **indirection through** a pointer to an incomplete type (other than *cv* `void`) ~~can be dereferenced~~ **is valid**. The lvalue thus obtained...

7. Change 8.10 [expr.eq] paragraph 2 as follows:

> ...Otherwise they compare equal if and only if they would refer to the same member of the same most derived object (4.5 [intro.object]) or the same subobject if ~~they were dereferenced~~ **indirection** with a hypothetical object of the associated class type **were performed**. [*Example:*...

8. Change 10.5 [dcl.link] paragraph 8:

> [*Note:* Because the language linkage is part of a function type, when **indirecting through** a pointer to C function ~~(for example) is dereferenced~~, the function to which ~~it~~ **the resulting lvalue** refers is considered a C function. —*end note*]

9. Change 11.3.2 [dcl.ref] paragraph 5 as follows:

> ...[*Note:* in particular, a null reference cannot exist in a well-defined program, because the only way to create such a reference would be to bind it to the "object" obtained by ~~dereferencing~~ **indirection through** a null pointer, which causes undefined behavior. As described...

10. Change 20.5.3.5 [allocator.requirements] table 27:

Variable	Definition
...	
c	a ~~dereferenceable~~ pointer of type `C*` **through which indirection is valid**
...	

11. Change 23.10.3.2 [pointer.traits.functions] as follows:

    Returns: The first member function returns a ~~dereferenceable~~ pointer to `r` obtained by calling `Ptr::pointer_to(r)` **through which indirection is valid**; an instantiation of this function is ill-formed...

12. Change 23.10.4 [util.dynamic.safety] paragraph 10 as follows:

    Effects: The `n` bytes starting at `p` no longer contain traceable pointer locations, independent of their type. Hence ~~pointers~~ **indirection through a pointer** located there ~~may not be dereferenced~~ **is undefined** if the object ~~they point~~ **it points** to was created by global `operator new` and not previously declared reachable...

13. Change 23.10.10 [specialized.algorithms] paragraph 1 as follows:

    ...is required to have the property that no exceptions are thrown from increment, assignment, comparison, or ~~dereference of~~ **indirection through** valid iterators...

14. Change 25.4.5.1.2 [locale.time.get.virtuals] paragraph 11 as follows:

    Requires: `t` shall ~~be dereferenceable~~ **point to an object**.

15. Change 26.4.4.2 [map.cons] paragraph 3 as follows:

    Requires: If the iterator's ~~dereference~~ **indirection** operator returns an lvalue or a const rvalue `pair<key_type, mapped_type>`, then both `key_type` and `mapped_type` shall be `CopyConstructible`.

16. Change 26.4.5.2 [multimap.cons] paragraph 3 as follows:

    Requires: If the iterator's ~~dereference~~ **indirection** operator returns an lvalue or a const rvalue `pair<key_type, mapped_type>`, then both `key_type` and `mapped_type` shall be `CopyConstructible`.

17. Change 26.4.6.2 [set.cons] paragraph 4 as follows:

    Requires: If the iterator's ~~dereference~~ **indirection** operator returns an lvalue or a non-const rvalue, then `Key` shall be `CopyConstructible`.

18. Change 26.4.7.2 [multiset.cons] paragraph 3 as follows:

    Requires: If the iterator's ~~dereference~~ **indirection** operator returns an lvalue or a const rvalue, then `Key` shall be `CopyConstructible`.

19. Change 27.5.3 [move.iterators] paragraph 1 as follows:

    Class template `move_iterator` is an iterator adaptor with the same behavior as the underlying iterator except that its ~~dereference~~ **indirection** operator implicitly converts the value returned by the underlying iterator's ~~dereference~~ **indirection** operator to an rvalue reference...

20. Change the title of 31.12.1.3 [re.regiter.deref] as follows:

    `regex_iterator` ~~**dereference**~~ **indirection**

21. Change the title of 31.12.2.3 [re.tokiter.deref] as follows:

    `regex_token_iterator` ~~**dereference**~~ **indirection**

1458. Address of incomplete type vs `operator&()`

Section: 8.3.1 [expr.unary.op] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2012-02-07

[Moved to DR at the October, 2012 meeting.]

According to 8.3.1 [expr.unary.op] paragraph 5,

> The address of an object of incomplete type can be taken, but if the complete type of that object is a class type that declares `operator&()` as a member function, then the behavior is undefined (and no diagnostic is required).

This should actually be "ill-formed, no diagnostic required" instead of undefined behavior, since the problem could be detected by whole-program analysis. Also, it's not clear what this means for constant expressions.

Proposed resolution (February, 2012):

Change 8.3.1 [expr.unary.op] paragraph 5 as follows:

~~The address of an object of incomplete type can be taken, but if the complete type of that object is a class type that declares `operator&()` as a member function, then the behavior is undefined (and no diagnostic is required).~~ **If `&` is applied to an lvalue of incomplete class type and the complete type declares `operator&()`, it is unspecified whether the operator has the built-in meaning or the operator function is called.** The operand of `&` shall not be a bit-field.

1553. `sizeof` **and xvalue bit-fields**

Section: 8.3.3 [expr.sizeof] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2012-09-13

[Moved to DR at the April, 2013 meeting.]

According to 8.3.3 [expr.sizeof] paragraph 1,

> The `sizeof` operator shall not be applied... to an lvalue that designates a bit-field.

Xvalues can also designate bit-fields and thus should presumably be addressed here as well.

Proposed resolution (October, 2012):

Change 8.3.3 [expr.sizeof] paragraph 1 as follows:

> The `sizeof` operator yields the number of bytes in the object representation of its operand. The operand is either an expression, which is an unevaluated operand (Clause 8 [expr]), or a parenthesized *type-id*. The `sizeof` operator shall not be applied to an expression that has function or incomplete type, to an enumeration type whose underlying type is not fixed before all its enumerators have been declared, to the parenthesized name of such types, or to ~~an lvalue~~ **a glvalue** that designates a bit-field. `sizeof(char)`**...**

---

292. Deallocation on exception in `new` **before arguments evaluated**

Section: 8.3.4 [expr.new] **Status:** CD3 **Submitter:** Andrei Iltchenko **Date:** 26 Jun 2001

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

According to the C++ Standard section 8.3.4 [expr.new] paragraph 21 it is unspecified whether the allocation function is called before evaluating the constructor arguments or after evaluating the constructor arguments but before entering the constructor.

On top of that paragraph 17 of the same section insists that

> If any part of the object initialization described above [Footnote: This may include evaluating a new-initializer and/or calling a constructor.] terminates by throwing an exception and a suitable deallocation function is found, the deallocation function is called to free the memory in which the object was being constructed... If no unambiguous matching deallocation function can be found, propagating the exception does not cause the object's memory to be freed...

Now suppose we have:

1. An implementation that always evaluates the constructor arguments first (for a new-expression that creates an object of a class type and has a new-initializer) and calls the allocation function afterwards.
2. A class like this:

```
struct  copy_throw  {
   copy_throw(const copy_throw&)
   {   throw  std::logic_error("Cannot copy!");   }
   copy_throw(long, copy_throw)
   {   }
   copy_throw()
   {   }
};
```

3. And a piece of code that looks like the one below:

```
int  main()
try  {
   copy_throw   an_object,     /* undefined behaviour */
      * a_pointer = ::new copy_throw(0, an_object);
   return  0;
}
catch(const std::logic_error&)
{   }
```

Here the new-expression '`::new copy_throw(0, an_object)`' throws an exception when evaluating the constructor's arguments and before the allocation function is called. However, 8.3.4 [expr.new] paragraph 17 prescribes that in such a case the implementation shall call the deallocation function to free the memory in which the object was being constructed, given that a matching deallocation function can be found.

So a call to the Standard library deallocation function '`::operator delete(void*)`' shall be issued, but what argument is an implementation supposed to supply to the deallocation function? As per 8.3.4 [expr.new] paragraph 17 - the argument is the address of the memory in

which the object was being constructed. Given that no memory has yet been allocated for the object, this will qualify as using an invalid pointer value, which is undefined behaviour by virtue of 6.7.4.2 [basic.stc.dynamic.deallocation] paragraph 4.

Suggested resolution:

Change the first sentence of 8.3.4 [expr.new] paragraph 17 to read:

> If the memory for the object being created has already been successfully allocated and any part of the object initialization described above...

Proposed resolution (March, 2008):

Change 8.3.4 [expr.new] paragraph 18 as follows:

> If any part of the object initialization described above [*Footnote:* ...] terminates by throwing an exception**, storage has been obtained for the object,** and a suitable deallocation function can be found, the deallocation function is called...

1464. Negative array bound in a *new-expression*

Section: 8.3.4 [expr.new] **Status:** CD3 **Submitter:** Mike Miller **Date:** 2012-02-12

[Accepted at the April, 2013 meeting.]

Currently, 8.3.4 [expr.new] paragraph 7 requires that an attempt to allocate an array with a negative length be diagnosed:

> If the value of that *expression* is less than zero or such that the size of the allocated object would exceed the implementation-defined limit, or if the *new-initializer* is a *braced-init-list* for which the number of *initializer-clause*s exceeds the number of elements to initialize, no storage is obtained and the *new-expression* terminates by throwing an exception of a type that would match a handler (18.3 [except.handle]) of type `std::bad_array_new_length` (21.6.3.2 [new.badlength]).

Checking for a negative bound will be lost, however, upon the adoption of paper N3323, as the *expression* will be converted to `std::size_t`, an unsigned type. Although the result of this conversion will likely also cause the check to fail (and will always do so when scaled by an element size larger than 1), it is not inconceivable that an implementation could provide a heap that capable of providing more than half the addressable range of `std::size_t`, and a request for a character array (with an element size of 1) with a negative bound close to `LONG_MIN` (assuming `std::size_t` is `unsigned long`) might actually succeed.

The wording of 8.3.4 [expr.new] paragraph 7 should be changed so that the test for a negative bound is applied to the value before conversion to `std::size_t`, or some other mechanism should be invented to preserve the check for a negative bound.

Additional note (August, 2012):

The goal for addressing this issue should be that an attempt to use an invalid bound (negative, greater than the maximum allowed, or more than the number implied by the initializer) will be ill-formed when the bound is a compile-time constant and will result in an exception otherwise.

Proposed resolution (October, 2012):

1. Change 6.9.2 [basic.compound] paragraph 2 as follows:

> These methods of constructing types can be applied recursively; restrictions are mentioned in 11.3.1 [dcl.ptr], 11.3.4 [dcl.array], 11.3.5 [dcl.fct], and 11.3.2 [dcl.ref]. **Constructing a type such that the number of bytes in its object representation exceeds the maximum value representable in the type `std::size_t` (21.2 [support.types]) is ill-formed.**

2. Change 8.3.4 [expr.new] paragraph 7 as follows:

> **The *expression* in a *noptr-new-declarator* is erroneous if:**
>
> - **the expression is of non-class type and its value before converting to `std::size_t` is less than zero;**
> - **the expression is of class type and its value before application of the second standard conversion (16.3.3.1.2 [over.ics.user]) [*Footnote:* If the conversion function returns a signed integer type, the second standard conversion converts to the unsigned type `std::size_t` and thus thwarts any attempt to detect a negative value afterwards. —*end footnote*] is less than zero;**
> - **its value is such that the size of the allocated object would exceed the implementation-defined limit (annex B [implimits]); or**
> - **the *new-initializer* is a *braced-init-list* and the number of array elements for which initializers are provided (including the terminating `'\0'` in a string literal (5.13.5 [lex.string])) exceeds the number of elements to initialize.**
>
> **If the *expression*, after converting to `std::size_t`, is a core constant expression and the expression is erroneous, the program is ill-formed. Otherwise, a *new-expression* with an erroneous expression does not call an allocation function and terminates by throwing an exception of a type that would match a handler (18.3 [except.handle]) of type `std::bad_array_new_length` (21.6.3.2 [new.badlength]).** When the value of the *expression* ~~in a *noptr-new-declarator*~~

is zero, the allocation function is called to allocate an array with no elements. ~~If the value of that *expression* is less than zero or such that the size of the allocated object would exceed the implementation-defined limit, or if the *new-initializer* is a *braced-init-list* for which the number of *initializer-clauses* exceeds the number of elements to initialize, no storage is obtained and the *new-expression* terminates by throwing an exception of a type that would match a handler (18.3 [except.handle]) of type `std::bad_array_new_length` (21.6.3.2 [new.badlength]).~~

(This resolution also resolves issue 1559.)

1559. String too long in initializer list of *new-expression*

Section: 8.3.4 [expr.new] **Status:** CD3 **Submitter:** John Spicer **Date:** 2012-09-21

[Moved to DR at the April, 2013 meeting.]

According to 8.3.4 [expr.new] paragraph 7,

> if the *new-initializer* is a *braced-init-list* for which the number of *initializer-clause*s exceeds the number of elements to initialize, no storage is obtained and the *new-expression* terminates by throwing an exception of a type that would match a handler (18.3 [except.handle]) of type `std::bad_array_new_length` (21.6.3.2 [new.badlength]).

This wording does not, but presumably should, require an exception to be thrown in a case like

```
void f() {
  int x = 3;
  new char[x]{"abc"};
}
```

(See also issue 1464.)

Proposed resolution (October, 2012):

This issue is resolved by the resolution of issue 1464.

1305. `alignof` applied to array of unknown size

Section: 8.3.6 [expr.alignof] **Status:** CD3 **Submitter:** Nikolay Ivchenkov **Date:** 2011-04-26

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

According to 8.3.6 [expr.alignof] paragraph 1,

> An `alignof` expression yields the alignment requirement of its operand type. The operand shall be a *type-id* representing a complete object type or an array thereof or a reference to a complete object type.

This (presumably unintentionally) excludes a reference to an array with an unknown bound but a complete element type; the bound is not needed to determine the alignment of the array.

Proposed resolution (August, 2011):

Change 8.3.6 [expr.alignof] paragraph 1 as follows:

> An `alignof` expression yields the alignment requirement of its operand type. The operand shall be a *type-id* representing a complete object type or an array thereof or a reference to ~~a complete object type~~ **one of those types**.

1354. Destructor exceptions for temporaries in noexcept expressions

Section: 8.3.7 [expr.unary.noexcept] **Status:** CD3 **Submitter:** Sebastian Redl **Date:** 2011-08-16

[Moved to DR at the October, 2012 meeting.]

The result of the `noexcept` operator does not consider possible exceptions thrown by the destructors for temporaries created in the operand expression.

Proposed resolution (February, 2012):

1. Change 4.6 [intro.execution] paragraph 10 as follows:

> A *full-expression* is an expression that is not a subexpression of another expression. **[*Note:* in some contexts such as unevaluated operands, a syntactic subexpression is considered a full-expression (Clause 8 [expr]). —*end note*]** If a language construct...

2. Change 8 [expr] paragraph 7 as follows:

> ...An unevaluated operand is not evaluated. **An unevaluated operand is considered a full-expression.** [*Note:*...

[Drafting note: This uniformly handles `sizeof(A())`, `noexcept(A())`, `typeid(A())`, *and* `decltype(A())` *with regard to the semantic requirements on* `~A` *(accessible and not deleted), which might be checked via SFINAE. A programmer can use* `decltype(new A)` *to avoid considering the destructor. If this is undesired, an alternative change just addresses the noexecept issue:]*

[Editing note: all the occurrences of "potentially evaluated" in 8.3.7 [expr.unary.noexcept] paragraph 3 should be hyphenated.]

1340. Complete type in member pointer expressions

Section: 8.5 [expr.mptr.oper] **Status:** CD3 **Submitter:** Daniel Krügler **Date:** 2011-08-10

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

Both the `.*` and `->*` operators (8.5 [expr.mptr.oper]) require that the class of the second operand be a complete object type. Current implementations do not enforce this requirement, and it is not clear that there is a need for it.

Proposed resolution (August, 2011):

1. Change 8.5 [expr.mptr.oper] paragraph 2 as follows:

> The binary operator `.*` binds its second operand, which shall be of type "pointer to member of `T`" ~~(where `T` is a completely-defined class type)~~ to its first operand...

2. Change 8.5 [expr.mptr.oper] paragraph 3 as follows:

> The binary operator `->*` binds its second operand, which shall be of type "pointer to member of `T`" ~~(where `T` is a completely-defined class type)~~ to its first operand...

1450. `INT_MIN % -1`

Section: 8.6 [expr.mul] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2012-01-31

[Moved to DR at the October, 2012 meeting.]

Issue 614 adopted the corresponding C99 wording for 8.6 [expr.mul] paragraph 4,

> ...if the quotient `a/b` is representable in the type of the result, `(a/b)*b + a%b` is equal to `a`.

in an attempt to ensure that `INT_MAX % -1` produces undefined behavior (because the result is not specified by the Standard). However, the new C draft makes the undefined behavior explicit:

> If the quotient `a/b` is representable, the expression `(a/b) * b + a%b` shall equal `a`; otherwise, the behavior of both `a/b` and `a%b` is undefined.

Should C++ adopt similar wording?

Proposed resolution (February, 2012):

Change 8.6 [expr.mul] paragraph 4 as follows:

> ...If the second operand of `/` or `%` is zero the behavior is undefined. For integral operands the `/` operator yields the algebraic quotient with any fractional part discarded;[81] if the quotient `a/b` is representable in the type of the result, `(a/b)*b + a%b` is equal to `a`**; otherwise, the behavior of both** `a/b` **and** `a%b` **is undefined**.

1504. Pointer arithmetic after derived-base conversion

Section: 8.7 [expr.add] **Status:** CD3 **Submitter:** Loïc Joly **Date:** 2012-05-20

[Moved to DR at the April, 2013 meeting.]

The current wording is not sufficiently clear that a pointer to a base class subobject of an array element cannot be used in pointer arithmetic.

Proposed resolution (October, 2012):

Add the following as a new paragraph before 8.7 [expr.add] paragraph 7:

For addition or subtraction, if the expressions `P` or `Q` have type "pointer to *cv* `T`", where `T` is different from the cv-unqualified array element type, the behavior is undefined. [*Note:* In particular, a pointer to a base class cannot be used for pointer arithmetic when the array contains objects of a derived class type. —*end note*]

If the value `0` is added to or subtracted...

1457. Undefined behavior in left-shift

Section: 8.8 [expr.shift] **Status:** CD3 **Submitter:** Howard Hinnant **Date:** 2012-02-04

[Moved to DR at the October, 2012 meeting.]

The current wording of 8.8 [expr.shift] paragraph 2 makes it undefined behavior to create the most-negative integer of a given type by left-shifting a (signed) `1` into the sign bit, even though this is not uncommonly done and works correctly on the majority of (twos-complement) architectures:

> ...if `E1` has a signed type and non-negative value, and $E1 \times 2^{E2}$ is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.

As a result, this technique cannot be used in a constant expression, which will break a significant amount of code.

Proposed resolution (February, 2012):

Change 8.8 [expr.shift] paragraph 2 as follows:

> ...if `E1` has a signed type and non-negative value, and $E1 \times 2^{E2}$ is representable in the **corresponding unsigned type of the** result type, then that **value, converted to the result type,** is the resulting value; otherwise, the behavior is undefined.

---

583. Relational pointer comparisons against the null pointer constant

Section: 8.9 [expr.rel] **Status:** CD3 **Submitter:** James Widman **Date:** 24 May 2006

[Moved to DR status at the April, 2013 meeting as paper N3624.]

In C, this is ill-formed (cf C99 6.5.8):

```
void f(char* s) {
    if (s < 0) { }
}
```

...but in C++, it's not. Why? Who would ever need to write `(s > 0)` when they could just as well write `(s != 0)`?

This has been in the language since the ARM (and possibly earlier); apparently it's because the pointer conversions (7.11 [conv.ptr]) need to be performed on both operands whenever one of the operands is of pointer type. So it looks like the "null-ptr-to-real-pointer-type" conversion is hitching a ride with the other pointer conversions.

Proposed resolution (April, 2013):

This issue is resolved by the resolution of issue 1512.

1512. Pointer comparison vs qualification conversions

Section: 8.9 [expr.rel] **Status:** CD3 **Submitter:** Steve Clamage **Date:** 2012-06-22

[Moved to DR status at the April, 2013 meeting as paper N3624.]

According to 8.9 [expr.rel] paragraph 2, describing pointer comparisons,

> Pointer conversions (7.11 [conv.ptr]) and qualification conversions (7.5 [conv.qual]) are performed on pointer operands (or on a pointer operand and a null pointer constant, or on two null pointer constants, at least one of which is non-integral) to bring them to their *composite pointer type*.

This would appear to make the following example ill-formed,

```
bool foo(int** x, const int** y) {
   return x < y;  // valid ?
}
```

because `int**` cannot be converted to `const int**`, according to the rules of 7.5 [conv.qual] paragraph 4. This seems too strict for pointer comparison, and current implementations accept the example.

Proposed resolution (November, 2012):

The proposed wording is found in document N3478.

(This resolution also resolves issue 583.)

1550. Parenthesized *throw-expression* operand of *conditional-expression*

Section: 8.16 [expr.cond] **Status:** CD3 **Submitter:** Nikolay Ivchenkov **Date:** 2012-09-04

[Moved to DR at the April, 2013 meeting.]

The current wording of 8.16 [expr.cond] paragraph 2 says,

> If either the second or the third operand has type `void`, then the lvalue-to-rvalue (7.1 [conv.lval]), array-to-pointer (7.2 [conv.array]), and function-to-pointer (7.3 [conv.func]) standard conversions are performed on the second and third operands, and one of the following shall hold:
>
> - The second or the third operand (but not both) is a *throw-expression* (18.1 [except.throw]); the result is of the type of the other and is a prvalue.
> - Both the second and the third operands have type `void`; the result is of type `void` and is a prvalue. [*Note:* This includes the case where both operands are *throw-expression*s. —*end note*]

A parenthesized *throw-expression* is a *primary-expression*, not a *throw-expression*. Should a parenthesized *throw-expression* be considered a *throw-expression* for this purpose?

Proposed resolution (October, 2012):

Change 8.16 [expr.cond] paragraph 2 as follows:

> If either the second or the third operand has type `void`, ~~then the lvalue-to-rvalue (7.1 [conv.lval]), array-to-pointer (7.2 [conv.array]), and function-to-pointer (7.3 [conv.func]) standard conversions are performed on the second and third operands, and~~ one of the following shall hold:
>
> - The second or the third operand (but not both) is a **(possibly parenthesized)** *throw-expression* (18.1 [except.throw]); the result is of the type **and value category** of the other ~~and is a prvalue~~.
> - ...

(This resolution also resolves issue 1560.)

1560. Gratuitous lvalue-to-rvalue conversion in *conditional-expression* with *throw-expression* operand

Section: 8.16 [expr.cond] **Status:** CD3 **Submitter:** Nikolay Ivchenkov **Date:** 2012-09-04

[Moved to DR at the April, 2013 meeting.]

A glvalue appearing as one operand of a *conditional-expression* in which the other operand is a *throw-expression* is converted to a prvalue, regardless of how the *conditional-expression* is used:

> If either the second or the third operand has type `void`, then the lvalue-to-rvalue (7.1 [conv.lval]), array-to-pointer (7.2 [conv.array]), and function-to-pointer (7.3 [conv.func]) standard conversions are performed on the second and third operands, and one of the following shall hold:
>
> - The second or the third operand (but not both) is a *throw-expression* (18.1 [except.throw]); the result is of the type of the other and is a prvalue.

This seems to be gratuitous and surprising.

Proposed resolution (October, 2012):

This issue is resolved by the resolution of issue 1550.

1527. Assignment from *braced-init-list*

Section: 8.18 [expr.ass] **Status:** CD3 **Submitter:** Mike Miller **Date:** 2012-07-23

[Moved to DR at the April, 2013 meeting.]

According to 8.18 [expr.ass] paragraph 9,

> A *braced-init-list* may appear on the right-hand side of
>
> - an assignment to a scalar...
> - an assignment defined by a user-defined assignment operator, in which case the initializer list is passed as the argument to the operator function.

Presumably the phrase "user-defined" is not intended to forbid an example like

```
struct A {
  A();
  A ( std::initializer_list<int> ) ;
};
void f() {
  A a;
  a = {37};
}
```

which relies on an implicitly-declared assignment operator.

Proposed resolution (August, 2012):

Change 8.18 [expr.ass] paragraph 9 as follows:

> A *braced-init-list* may appear on the right-hand side of
>
> - an assignment to a scalar, in which case the initializer list shall have at most a single element. The meaning of `x={v}`, where `T` is the scalar type of the expression `x`, is that of `x=T(v)` except that no narrowing conversion (8.5.4) is allowed. The meaning of `x={}` is `x=T()`.
> - an assignment ~~defined by a user-defined assignment operator~~ **to an object of class type**, in which case the initializer list is passed as the argument to the **assignment** operator function **selected by overload resolution (16.5.3 [over.ass], 16.3 [over.match])**.

1538. C-style cast in *braced-init-list* assignment

Section: 8.18 [expr.ass] **Status:** CD3 **Submitter:** Daniel Krügler **Date:** 2012-08-14

[Moved to DR at the April, 2013 meeting.]

According to 8.18 [expr.ass] paragraph 9,

> The meaning of `x={v}`, where `T` is the scalar type of the expression `x`, is that of `x=T(v)` except that no narrowing conversion (11.6.4 [dcl.init.list]) is allowed. The meaning of `x={}` is `x=T()`.

This definition adds a gratuitous C-style cast to the right-hand operand, inadvertently allowing such things as base-to-derived conversions and circumvention of access checking.

Proposed resolution (October, 2012):

Change 8.18 [expr.ass] paragraph 9 as follows:

> The meaning of `x={v}`, where `T` is the scalar type of the expression `x`, is that of ~~`x=T(v)` except that no narrowing conversion (11.6.4 [dcl.init.list]) is allowed~~ `x=T{v}`. The meaning of `x={}` is ~~`x=T()`~~ `x=T{}`.

1264. Use of `this` in `constexpr` constructor

Section: 8.20 [expr.const] **Status:** CD3 **Submitter:** Jason Merrill **Date:** 2011-03-18

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

Proposed resolution (August, 2011):

This issue is resolved by the resolution of issue 1369.

1293. String literals in constant expressions

Section: 8.20 [expr.const] **Status:** CD3 **Submitter:** Daniel Krügler **Date:** 2011-04-11

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

It is not clear whether a string literal can be used in a constant expression.

Proposed resolution (August, 2011):

Change 8.20 [expr.const] paragraph 2 as follows:

- an lvalue-to-rvalue conversion (7.1 [conv.lval]) unless it is applied to
  - a glvalue of integral or enumeration type that refers to a non-volatile const object with a preceding initialization, initialized with a constant expression **[*Note:* a string literal (5.13.5 [lex.string]) corresponds to an array of such objects. —*end note*]**, or
  - ...

1311. Volatile lvalues in constant expressions

Section: 8.20 [expr.const] **Status:** CD3 **Submitter:** Nikolay Ivchenkov **Date:** 2011-05-06

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

The current wording of 8.20 [expr.const] paragraph 2 does not, but should, prohibit use of volatile glvalues in constant expressions.

Proposed resolution (August, 2011):

Change 8.20 [expr.const] paragraph 2 as follows:

- an lvalue-to-rvalue conversion (7.1 [conv.lval]) unless it is applied to
  - a **non-volatile** glvalue of integral or enumeration type that refers to a non-volatile const object with a preceding initialization, initialized with a constant expression, or
  - a **non-volatile** glvalue of literal type that refers to a non-volatile object defined with `constexpr`, or that refers to a sub-object of such an object, or
  - a **non-volatile** glvalue of literal type that refers to a non-volatile temporary object whose lifetime has not ended, initialized with a constant expression;

1312. Simulated `reinterpret_cast` in constant expressions

Section: 8.20 [expr.const] **Status:** CD3 **Submitter:** Nikolay Ivchenkov **Date:** 2011-05-06

[Moved to DR at the October, 2012 meeting.]

Although a `reinterpret_cast` is prohibited in a constant expression, casting to and from `void*` can achieve the same effect.

Proposed resolution (August, 2011):

Change 8.20 [expr.const] paragraph 2 as follows:

- an lvalue-to-rvalue conversion (7.1 [conv.lval]) unless it is applied to...
- **an lvalue-to-rvalue conversion (7.1 [conv.lval]) that is applied to a glvalue of type *cv1* `T` that refers to an object of type *cv2* `U`, where `T` and `U` are neither the same type nor similar types (7.5 [conv.qual]);**
- an lvalue-to-rvalue conversion (7.1 [conv.lval]) that is applied to a glvalue that refers to a non-active member...

Note, January, 2012:

Additional discussion has occurred, so this issue has been returned to "review" status to allow further consideration.

Proposed resolution (February, 2012):

Change 8.20 [expr.const] paragraph 2 as follows:

- ...
- an *id-expression* that refers to a variable or data member of reference type unless...
- **a conversion from type *cv* `void *` to a pointer-to-object type;**
- a dynamic cast (8.2.7 [expr.dynamic.cast]);
- ...

1313. Undefined pointer arithmetic in constant expressions

Section: 8.20 [expr.const] **Status:** CD3 **Submitter:** Jens Maurer **Date:** 2011-05-07

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

The requirements for constant expressions do not currently, but should, exclude expressions that have undefined behavior, such as pointer arithmetic when the pointers do not point to elements of the same array.

Proposed resolution (August, 2011):

Change 8.20 [expr.const] paragraph 2 as follows:

- ...
- ~~a result that is not mathematically defined or not in the range of representable values for its type;~~
- **an operation that would have undefined behavior [*Note:* including, for example, signed integer overflow (Clause 8 [expr]), certain pointer arithmetic (8.7 [expr.add]), division by zero (8.6 [expr.mul]), or certain shift operations (8.8 [expr.shift]) — *end note*];**
- ...
- ~~a subtraction (8.7 [expr.add]) where both operands are pointers;~~
- ...

1364. `constexpr` **function parameters**

Section: 8.20 [expr.const] **Status:** CD3 **Submitter:** Sean Hunt **Date:** 2011-08-17

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

Use of a parameter in a `constexpr` function appears to be ill-formed, because the lvalue-to-rvalue conversion on the parameter is not one of those permitted in a constant expression.

Proposed resolution (August, 2011):

1. Change the indicated bullet of 8.20 [expr.const] paragraph 2 as follows:
   - an invocation of a `constexpr` constructor with arguments that, when substituted by function invocation substitution (10.1.5 [dcl.constexpr]), do not produce all constant expressions for the constructor calls and full-expressions in the *mem-initializer*s **(including conversions)**; [*Example:*...
2. Delete the final bullet of 10.1.5 [dcl.constexpr] paragraph 3 and move the deleted "." to the preceding sub-bullet:
   - ~~every constructor call and implicit conversion used in initializing the return value (9.6.3 [stmt.return], 11.6 [dcl.init]) shall be one of those allowed in a constant expression (8.20 [expr.const]).~~
3. Delete the final bullet of 10.1.5 [dcl.constexpr] paragraph 4 and change the preceding bullet as follows:
   - every *assignment-expression* that is an *initializer-clause* appearing directly or indirectly within a *brace-or-equal-initializer* for a non-static data member that is not named by a *mem-initializer-id* shall be a constant expression~~; and~~**.**
   - ~~every implicit conversion used in converting a constructor argument to the corresponding parameter type and converting a full-expression to the corresponding member type shall be one of those allowed in a constant expression.~~

1365. Calling undefined `constexpr` **functions**

Section: 8.20 [expr.const] **Status:** CD3 **Submitter:** Sean Hunt **Date:** 2011-08-17

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

The current wording appears to allow calling a `constexpr` function that is never defined within the body of a `constexpr` function. (The wording was intended to allow mutually-recursive `constexpr` functions but require that the not-yet-defined function be defined before it would be needed in an actual constant expression.)

Proposed resolution (August, 2011):

Change the indicated bullet of 8.20 [expr.const] paragraph 2 as follows:

- an invocation of an undefined `constexpr` function or an undefined `constexpr` constructor ~~outside the definition of a `constexpr` function or a `constexpr` constructor~~;

1367. Use of `this` in a constant expression

Section: 8.20 [expr.const] **Status:** CD3 **Submitter:** Jason Merrill **Date:** 2011-08-17

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

The provisions allowing the use of `this` in a constant expression appear to be unnecessary, as any uses of `this` in a constant expression that are valid will be replaced by function invocation substitution.

Proposed resolution (August, 2011):

This issue is resolved by the resolution of issue 1369.

1454. Passing constants through `constexpr` functions via references

Section: 8.20 [expr.const] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2011-12-27

[Moved to DR at the October, 2012 meeting.]

The current wording incorrectly appears to make the following example ill-formed:

```
constexpr const int &f(const int &n) { return n; }
constexpr int k = f(0);   // ill-formed
```

Proposed resolution (February, 2012):

1. Change 8.20 [expr.const] paragraph 2 as follows:

   A *conditional-expression* is a *core constant expression* unless it involves one of the following...

   - ...
   - an invocation of a `constexpr` function with arguments that, when substituted by function invocation substitution (10.1.5 [dcl.constexpr]), do not produce a **core** constant expression; [*Example:*...
   - an invocation of a `constexpr` constructor with arguments that, when substituted by function invocation substitution (10.1.5 [dcl.constexpr]), do not produce all **core** constant expressions for the constructor calls and full-expressions in the *mem-initializer*s; [*Example:*...
   - ...
   - an lvalue-to-rvalue conversion (7.1 [conv.lval]) unless it is applied to
     - ...
     - a glvalue of literal type that refers to a non-volatile temporary object whose lifetime has not ended, initialized with a **core** constant expression;
   - ...
   - an *id-expression* that refers to a variable or data member of reference type unless the reference has a preceding initialization~~, initialized with a constant expression;~~ **and either**
     - **it is initialized with a constant expression or**
     - **it is a non-static data member of a temporary object whose lifetime has not ended and is initialized with a core constant expression;**
   - ...

2. Change 8.20 [expr.const] paragraph 3 as follows, dividing it into two paragraphs:

   ~~A *literal constant expression* is a prvalue core constant expression of literal type, but not pointer type.~~ An *integral constant expression* is ~~a literal constant~~ **an** expression of integral or unscoped enumeration type**, implicitly converted to a prvalue, where the converted expression is a core constant expression**. [*Note:* Such expressions may be used as array bounds (11.3.4 [dcl.array], 8.3.4 [expr.new]), as bit-field lengths (12.2.4 [class.bit]), as enumerator initializers if the underlying type is not fixed (10.2 [dcl.enum]), as null pointer constants (7.11 [conv.ptr]), and as alignments (10.6.2 [dcl.align]). —*end note*] A *converted constant expression* of type `T` is ~~a literal constant~~ **an** expression, implicitly converted to **a prvalue of** type `T`, where the ~~implicit conversion (if any) is permitted in a literal~~ **converted expression is a core** constant expression and the implicit conversion sequence contains only user-defined conversions, lvalue-to-rvalue conversions (7.1 [conv.lval]), integral promotions (7.6 [conv.prom]), and integral conversions (7.8 [conv.integral]) other

than narrowing conversions (11.6.4 [dcl.init.list]). [*Note:* such expressions may be used as case expressions (9.4.2 [stmt.switch]), as enumerator initializers if the underlying type is fixed (10.2 [dcl.enum]), and as integral or enumeration non-type template arguments (17.3 [temp.arg]). —*end note*]

A *literal constant expression* is a prvalue core constant expression of literal type, but not pointer type (after conversions as required by the context). For a literal constant expression of array or class type, each subobject of its value shall have been initialized by a constant expression. A *reference constant expression* is an lvalue core constant expression that designates an object with static storage duration or a function. An *address constant expression* is a prvalue core constant expression **(after conversions as required by the context) of type** `std::nullptr_t` **or** of pointer type that evaluates to the address of an object with static storage duration, to the address of a function, or to a null pointer value~~, or a prvalue core constant expression of type `std::nullptr_t`~~. Collectively, literal constant expressions, reference constant expressions, and address constant expressions are called *constant expressions*.

3. Change the second example 10.1.5 [dcl.constexpr] paragraph 5 as follows:

```
constexpr int f(bool b)
  { return b ? throw 0 : 0; }                  // OK
constexpr int f() { throw 0 return f(true); }  // ill-formed, no diagnostic required
...
```

This resolution also resolves issue 1455.

1455. Lvalue converted constant expressions

Section: 8.20 [expr.const] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2012-01-14

[Moved to DR at the October, 2012 meeting.]

A "converted constant expression" must be a literal constant expression, which is a prvalue, and thus can't be an lvalue. This is unintended; the lvalue-to-rvalue conversion should be applied as necessary.

Proposed resolution (February, 2012):

This issue is resolved by the resolution of issue 1454.

1456. Address constant expression designating the one-past-the-end address

Section: 8.20 [expr.const] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2012-01-14

[Moved to DR at the April, 2013 meeting.]

Currently an address constant expression cannot designate the address one past the end of an array. This seems unfortunate.

Proposed resolution (August, 2012):

Change 8.20 [expr.const] paragraph 3 as follows:

> ...An *address constant expression* is a prvalue core constant expression of pointer type that evaluates to the address of an object with static storage duration, **to the address one past the last element of an array with static storage duration,** to the address of a function, or to a null pointer value, or a prvalue core constant expression of type `std::nullptr_t`...

1480. Constant initialization via non-constant temporary

Section: 8.20 [expr.const] **Status:** CD3 **Submitter:** Daniel Krügler **Date:** 2012-03-17

The following initializations appear to be well-formed:

```
struct C {
 int m;
 constexpr C(int m ) : m{m} {};
 constexpr int get() { return m; };
};

C&& rr = C{1};
constexpr int k1 = rr.get();

const C& c1 = C{1};
constexpr int k2 = c1.get();
```

They appear to fall under the bullet of 8.20 [expr.const] paragraph 2,

- an lvalue-to-rvalue conversion (7.1 [conv.lval]) unless it is applied to

- ...
- a non-volatile glvalue of literal type that refers to a non-volatile temporary object whose lifetime has not ended, initialized with a constant expression;

The problem in this example is that the referenced temporary object is not a constant, so it would be well-defined for intervening code to modify its value before it was used in the later initialization.

Additional note (February, 2013):

The intent is that non-const references to temporaries should be allowed within constant expressions, e.g., across `constexpr` function calls, but not as the result of a constant expression.

Proposed resolution (April, 2013):

This issue was rendered moot by paper N3652, adopted at the April, 2013 meeting.

1535. `typeid` **in core constant expressions**

Section: 8.20 [expr.const] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2012-08-10

[Moved to DR at the April, 2013 meeting.]

One of the criteria in 8.20 [expr.const] paragraph 2 for disqualifying an expression from being a constant expression is:

- a typeid expression (8.2.8 [expr.typeid]) whose operand is of a polymorphic class type;

on the basis that a runtime test for the dynamic type is inconsistent with a constant expression. However, it is only glvalues of polymorphic type that require a runtime test; *type-id*s and prvalues with a polymorphic type could (and should) be permitted in constant expressions.

Proposed resolution (October, 2012):

Change 8.20 [expr.const] paragraph 2 as follows:

- ...
- a typeid expression (8.2.8 [expr.typeid]) whose operand is **a glvalue** of a polymorphic class type;

1537. Optional compile-time evaluation of constant expressions

Section: 8.20 [expr.const] **Status:** CD3 **Submitter:** John Spicer **Date:** 2012-08-14

[Moved to DR at the April, 2013 meeting.]

According to the note in 8.20 [expr.const] paragraph 4,

> [*Note:* Although in some contexts constant expressions must be evaluated during program translation, others may be evaluated during program execution. Since this International Standard imposes no restrictions on the accuracy of floating-point operations, it is unspecified whether the evaluation of a floating-point expression during translation yields the same result as the evaluation of the same expression (or the same operations on the same values) during program execution.

With the advent of narrowing rules, which require the compiler to evaluate constant expressions in more contexts than was the case in C++03 in order to determine whether an expression is well formed or not, this wording is not sufficiently clear in stating that even in cases where the computation must be done at compile time, the implementation is free to use the result of a runtime calculation rather than preserving the one computed at compile time.

Proposed resolution (October, 2012):

Change 8.20 [expr.const] paragraph 4 as follows:

> [*Note:* ~~Although in some contexts constant expressions must be evaluated during program translation, others may be evaluated during program execution.~~ Since this International Standard imposes no restrictions on the accuracy of floating-point operations, it is unspecified whether the evaluation of a floating-point expression during translation yields the same result as the evaluation of the same expression (or the same operations on the same values) during program execution. [*Footnote:* Nonetheless, implementations are encouraged to provide consistent results, irrespective of whether the evaluation was ~~actually~~ performed during translation **and/**or during program execution. *—end footnote*] [*Example:*...

---

631. Jumping into a "then" clause

Section: 9.4.1 [stmt.if] **Status:** CD3 **Submitter:** James Kanze **Date:** 24 April 2007

[Moved to DR at the October, 2012 meeting.]

9.4.1 [stmt.if] is silent about whether the `else` clause of an `if` statement is executed if the condition is not evaluated. (This could occur via a `goto` or a `longjmp`.) C99 covers the `goto` case with the following provision:

> If the first substatement is reached via a label, the second substatement is not executed.

It should probably also be stated that the condition is not evaluated when the "then" clause is entered directly.

Proposed resolution (February, 2012):

Change 9.4.1 [stmt.if] paragraph 1 as follows:

> If the condition (9.4 [stmt.select]) yields `true` the first substatement is executed. If the `else` part of the selection statement is present and the condition yields `false`, the second substatement is executed. **If the first substatement is reached via a label, the condition is not evaluated and the second substatement is not executed.** In the second form...

1442. Argument-dependent lookup in the range-based `for`

Section: 9.5.4 [stmt.ranged] **Status:** CD3 **Submitter:** Mike Miller **Date:** 2012-01-16

[Moved to DR at the April, 2013 meeting.]

It is not clear whether the reference to argument-dependent lookup in 9.5.4 [stmt.ranged] paragraph 1 bullet 3 should be "pure" argument-dependent lookup (with no unqualified name lookup component) or the usual lookup that is invoked when argument-dependent lookup is done, i.e., unqualified lookup, potentially augmented by the associated namespaces.

Proposed resolution (October, 2012):

Change 9.5.4 [stmt.ranged] paragraph 1 bullet 3 as follows:

- otherwise, *begin-expr* and *end-expr* are `begin(__range)` and `end(__range)`, respectively, where `begin` and `end` are looked up ~~with argument-dependent lookup~~ **in the associated namespaces** (6.4.2 [basic.lookup.argdep]). ~~For the purposes of this name lookup, namespace `std` is an associated namespace.~~ **[*Note:* Ordinary unqualified lookup (6.4.1 [basic.lookup.unqual]) is not performed. — *end note*]**

1541. *cv* `void` return types

Section: 9.6.3 [stmt.return] **Status:** CD3 **Submitter:** Sean Hunt **Date:** 2012-08-21

[Moved to DR at the April, 2013 meeting.]

According to 9.6.3 [stmt.return] paragraph 3,

> A return statement with neither an expression nor a *braced-init-list* can be used only in functions that do not return a value, that is, a function with the return type `void`, a constructor (15.1 [class.ctor]), or a destructor (15.4 [class.dtor]).

However, paragraph 3 allows a return type of *cv* `void` in cases where the expression in the `return` statement has type `void`. Should paragraph 2 follow suit?

Proposed resolution (October, 2012):

Change 9.6.3 [stmt.return] paragraph 2 as follows:

> A return statement with neither an *expression* nor a *braced-init-list* can be used only in functions that do not return a value, that is, a function with the return type ***cv*** `void`, a constructor (15.1 [class.ctor]), or a destructor (15.4 [class.dtor]). A return statement with an expression of non-void type...

1544. Linkage of member of unnamed namespace

Section: 10.1.1 [dcl.stc] **Status:** CD3 **Submitter:** Nikolay Ivchenkov **Date:** 2012-08-24

[Moved to DR at the April, 2013 meeting.]

There is a contradiction in the specification of the linkage of members of the unnamed namespace, for example

```
namespace {
  int x;
}
```

According to 6.5 [basic.link] paragraph 4,

> An unnamed namespace or a namespace declared directly or indirectly within an unnamed namespace has internal linkage. All other namespaces have external linkage. A name having namespace scope that has not been given internal linkage above has the same linkage as the enclosing namespace if it is the name of
>
> - a variable; or
> - ...

which would give `x` internal linkage. However, 10.1.1 [dcl.stc] paragraph 7 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`.

This would give `x` external linkage. (This appears to have been a missed edit from the resolution of issue 1113.)

Proposed resolution (October, 2012):

Delete 10.1.1 [dcl.stc] paragraph 7:

> ~~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.~~

1437. `alignas` in *alias-declaration*

Section: 10.1.3 [dcl.typedef] **Status:** CD3 **Submitter:** Daveed Vandevoorde **Date:** 2012-01-02

[Moved to DR at the April, 2013 meeting.]

Consider the following:

```
struct S { int i; };
using A alignas(alignof(long long)) = S;
```

10.6.2 [dcl.align] paragraph 1 allows an *alignment-specifier* to be applied to the declaration of a class or enumeration type, which `A` arguably is. The specification should be clarified to indicate that such a usage is not permitted, however.

Proposed resolution (October, 2012):

Change 10.6.2 [dcl.align] paragraph 1 as follows:

> An *alignment-specifier* may be applied to a variable or to a class data member, but it shall not be applied to a bit-field, a function parameter, ~~the formal parameter of a catch clause~~ **an *exception-declaration*** (18.3 [except.handle]), or a variable declared with the `register` storage class specifier. An *alignment-specifier* may also be applied to the declaration ~~of a class or enumeration type~~ **or definition of a class (in an *elaborated-type-specifier* (10.1.7.3 [dcl.type.elab]) or *class-head* (Clause 12 [class]), respectively) and to the declaration or definition of an enumeration (in an *opaque-enum-declaration* or *enum-head*, respectively (10.2 [dcl.enum]))**. An *alignment-specifier* with an ellipsis is a pack expansion (17.6.3 [temp.variadic]).

1358. Unintentionally ill-formed `constexpr` function template instances

Section: 10.1.5 [dcl.constexpr] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2011-08-16

[Moved to DR status at the April, 2013 meeting.]

The permission granted implementations in 10.1.5 [dcl.constexpr] paragraph 5 to diagnose definitions of `constexpr` functions that can never be used in a constant expression should not apply to an instantiated `constexpr` function template.

Notes from the August, 2011 meeting:

The CWG also decided to treat the following example under this issue, although it does not involve a function template:

```
int f();   // not constexpr
struct A {
  int m;
  constexpr A(int i = f()) : m(i) { }
};
struct B {
  A a;
} b;
```

This is ill-formed, no diagnostic required, because the defaulted default constructor of `B` will be declared `constexpr` but can never be invoked in a constant expression. See issue 1360.

Proposed resolution (February, 2012):

1. Change 10.1.5 [dcl.constexpr] paragraph 5 as follows:

> ... *—end example*]
>
> For a **non-template, non-defaulted** `constexpr` function, if no function argument values exist such that the function invocation substitution would produce a constant expression (8.20 [expr.const]), the program is ill-formed; no diagnostic required. For a **non-template, non-defaulted, non-inheriting** `constexpr` constructor, if no argument values exist such that after function invocation substitution, every constructor call and full-expression in the *mem-initializer*s would be a constant expression (including conversions), the program is ill-formed; no diagnostic required. **For a `constexpr` function template or member function of a class template, if no instantiation would be well-formed when considered as a non-template `constexpr` function, the program is ill-formed; no diagnostic required.** [*Example:*...

2. Delete 10.1.5 [dcl.constexpr] paragraph 6:

> ~~If the instantiated template specialization of a `constexpr` function template or member function of a class template would fail to satisfy the requirements for a `constexpr` function or `constexpr` constructor, that specialization is not a `constexpr` function or `constexpr` constructor. [*Note:* If the function is a member function it will still be `const` as described below. *—end note*] If no specialization of the template would yield a `constexpr` function or `constexpr` constructor, the program is ill-formed; no diagnostic required.~~

Additional notes, February, 2012:

The proposed resolution inadvertently removes the provision allowing specializations of `constexpr` templates to violate the requirements of 10.1.5 [dcl.constexpr]. It is being retained in "drafting" status pending additional work.

Proposed resolution (October, 2012):

Change 10.1.5 [dcl.constexpr] paragraphs 5-6 as follows:

> ...For a **non-template, non-defaulted** `constexpr` function, if no function argument values exist such that the function invocation substitution would produce a constant expression (8.20 [expr.const]), the program is ill-formed; no diagnostic required. For a **non-template, non-defaulted, non-inheriting** `constexpr` constructor, if no argument values exist such that after function invocation substitution, every constructor call and full-expression in the *mem-initializer*s would be a constant expression (including conversions), the program is ill-formed; no diagnostic required. [*Example:* ... *—end example*]
>
> If the instantiated template specialization of a `constexpr` function template or member function of a class template would fail to satisfy the requirements for a `constexpr` function or `constexpr` constructor, that specialization is ~~not~~ **still** a `constexpr` function or `constexpr` constructor**, even though a call to such a function cannot appear in a constant expression**. ~~[*Note:* If the function is a member function it will still be `const` as described below. *—end note*]~~ If no specialization of the template would ~~yield~~ **satisfy the requirements for** a `constexpr` function or `constexpr` constructor **when considered as a non-template function or constructor**, the ~~program~~ **template** is ill-formed; no diagnostic required.

Additional note (January, 2013):

Questions arose in the discussion of issue 1581 as to whether this approach — making the specialization of a `constexpr` function template or member function of a class template still `constexpr` but unable to be invoked in a constant context — is correct. The implication is that class types might be categorized as literal but not be able to be instantiated at compile time. This issue is therefore returned to "review" status to allow further consideration of this question.

1359. `constexpr` union constructors

Section: 10.1.5 [dcl.constexpr] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2011-08-16

[Moved to DR at the October, 2012 meeting.]

A `constexpr` constructor is required to initialize all non-static data members (10.1.5 [dcl.constexpr] paragraph 4), which conflicts with the requirement that a constructor for a union is permitted to initialize only a single non-static data member (15.6.2 [class.base.init] paragraph 8).

Proposed resolution (February, 2012):

Change 10.1.5 [dcl.constexpr] paragraph 4 as follows:

In a definition of a `constexpr` constructor, each of the parameter types shall be a literal type. In addition, either its *function-body* shall be `= delete` or `= default` or it shall satisfy the following constraints:

- ...
- every **non-variant** non-static data member and base class sub-object shall be initialized (15.6.2 [class.base.init]);
- **if the class is a non-empty union, or for each non-empty anonymous union member of a non-union class, exactly one non-static data member shall be initialized;**
- ...

1366. Deleted `constexpr` constructors and virtual base classes

Section: 10.1.5 [dcl.constexpr] **Status:** CD3 **Submitter:** Sean Hunt **Date:** 2011-08-17

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

The requirement that a class with a `constexpr` constructor cannot have a virtual base only applies to constructors with non-deleted and non-defaulted *function-body*s. This seems like an oversight.

Proposed resolution (August, 2011):

Change 10.1.5 [dcl.constexpr] paragraph 4 as follows:

> ~~In a~~ **The** definition of a `constexpr` constructor~~, each of the parameter types shall be a literal type. In addition, either its *function-body* shall be = `delete` or = `default` or it~~ shall satisfy the following constraints:
>
> - the class shall not have any virtual base classes;
> - **each of the parameter types shall be a literal type;**
> - its *function-body* shall not be a *function-try-block*;
>
> **In addition, either its *function-body* shall be** = `delete` **or it shall satisfy the following constraints:**
>
> - **either its *function-body* shall be** = `default` `or` the compound-statement of its function-body shall contain only...

1369. Function invocation substitution of `this`

Section: 10.1.5 [dcl.constexpr] **Status:** CD3 **Submitter:** Jens Maurer **Date:** 2011-08-18

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

Function invocation substitution (10.1.5 [dcl.constexpr] paragraph 5) seems underspecified with respect to `this`.

Proposed resolution (August, 2011):

1. Change the indicated bullet of 8.20 [expr.const] paragraph 2 as follows:

   - `this` (~~8.1 [expr.prim]~~ **_N4567_.5.1.1 [expr.prim.general]**) ~~unless it appears as the *postfix-expression* in a class member access expression, including the result of the implicit transformation in the body of a non-static member function (12.2.2 [class.mfct.non-static])~~ **[*Note:* when evaluating a constant expression, function invocation substitution (10.1.5 [dcl.constexpr]) replaces each occurrence of `this` in a `constexpr` member function with a pointer to the class object. —*end note*]**;

2. Change 10.1.5 [dcl.constexpr] paragraph 5 as follows (converting the running text into a bulleted list):

   > *Function invocation substitution* for a call of a *constexpr* function or of a *constexpr* constructor means**:**
   >
   > - implicitly converting each argument to the corresponding parameter type as if by copy-initialization,[91]
   > - substituting that converted expression for each use of the corresponding parameter in the *function-body*,
   > - **in a member function, substituting for each use of `this` (12.2.2.1 [class.this]) a prvalue pointer whose value is the address of the object for which the member function is called,** and
   > - ~~for~~ **in a `constexpr` function~~s~~**,** implicitly converting the resulting returned expression or *braced-init-list* to the return type of the function as if by copy-initialization.
   >
   > Such substitution...

This resolution also resolves issues 1264 and 1367.

1597. Misleading `constexpr` example

Section: 10.1.5 [dcl.constexpr] **Status:** CD3 **Submitter:** John Spicer **Date:** 2012-12-21

[Addressed by the adoption of paper N3652 at the April, 2013 meeting.]

One of the examples in 10.1.5 [dcl.constexpr] paragraph 3 reads,

```
constexpr int prev(int x)
  { return --x; }       // error: use of decrement
```

According to paragraph 5, this ill-formed, no diagnostic required:

> For a constexpr function, if no function argument values exist such that the function invocation substitution would produce a constant expression (8.20 [expr.const]), the program is ill-formed; no diagnostic required.

However, the surrounding errors in the example have required diagnostics, potentially leading the reader to the mistaken conclusion that this error must be diagnosed as well. The example should be removed or the comment updated to reflect its true status.

Proposed resolution (June, 2013):

This issue is no longer relevant after the adoption of the changes to `constexpr` in paper N3652.

---

539. Constraints on *type-specifier-seq*

Section: 10.1.7 [dcl.type] **Status:** CD3 **Submitter:** Mike Miller **Date:** 5 October 2005

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

The constraints on *type-specifier*s given in 10.1.7 [dcl.type] paragraphs 2 and 3 (at most one *type-specifier* except as specified, at least one *type-specifier*, no redundant cv-qualifiers) are couched in terms of *decl-specifier-seq*s and *declaration*s. However, they should also apply to constructs that are not syntactically *declaration*s and that are defined to use *type-specifier-seq*s, including 8.3.4 [expr.new], 9.6 [stmt.jump], 11.1 [dcl.name], and 15.3.2 [class.conv.fct].

Proposed resolution (August, 2011):

Change 10.1.7 [dcl.type] paragraph 3 as follows:

> ~~At~~ **Except in a declaration of a constructor, destructor, or conversion function, at** least one *type-specifier* that is not a *cv-qualifier* ~~is required in a declaration unless it declares a constructor, destructor or conversion function~~ **shall appear in a complete *type-specifier-seq* or a complete *decl-specifier-seq***.[92] A *type-specifier-seq* shall not define...

(Note: paper N2546, voted into the Working Draft in February, 2008, addresses part of this issue.)

1265. Mixed use of the `auto` specifier

Section: 10.1.7.4 [dcl.spec.auto] **Status:** CD3 **Submitter:** Michael Wong **Date:** 2011-03-20

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

The current wording of 10.1.7.4 [dcl.spec.auto] does not appear to forbid using the `auto` specifier for both a function declaration with a trailing return type and a variable definition in the same declaration, e.g.,

```
auto f() -> int, i = 0;
```

(See also issue 1347.)

Proposed resolution (August, 2011):

Change 10.1.7.4 [dcl.spec.auto] paragraph 7 as follows:

> If the list of declarators contains more than one declarator, **they shall all form declarations of variables. The** ~~the~~ type of each declared variable is determined as described above~~. If~~**, and if** the type deduced for the template parameter `U` is not the same in each deduction, the program is ill-formed. [*Example:*...

1346. *expression-list* initializers and the `auto` specifier

Section: 10.1.7.4 [dcl.spec.auto] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2011-08-16

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

10.1.7.4 [dcl.spec.auto] does not address the case when the initializer for an `auto` variable is a parenthesized *expression-list*.

Proposed resolution (August, 2011):

Change 10.1.7.4 [dcl.spec.auto] paragraph 3 as follows:

...`auto` shall appear as one of the *decl-specifier*s in the *decl-specifier-seq* and the *decl-specifier-seq* shall be followed by one or more *init-declarator*s, each of which shall have a non-empty initializer. **In an *initializer* of the form**

(***expression-list***)

the *expression-list* shall be a single *assignment-expression*. [*Example:*...

1347. Consistency of `auto` in multiple-declarator declarations

Section: 10.1.7.4 [dcl.spec.auto] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2011-08-16

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

The intent of 10.1.7.4 [dcl.spec.auto] paragraph 7 appears to have been that the type represented by `auto` should be the same for each declarator in the declaration. However, the current wording does not achieve that goal. For example, in

```
auto a = 0, b = { 1, 2, 2 };
```

the `auto` specifier represents `int` in the first declarator and `std::initializer_list<int>` in the second. (See also issue 1265.)

Proposed resolution (August, 2011):

Move the example in 10.1.7.4 [dcl.spec.auto] paragraph 7 into that of paragraph 6 and change paragraph 7 as follows:

...[*Example:*

```
auto x1 = { 1, 2 };   // decltype(x1) is std::initializer_list<int>
auto x2 = { 1, 2.0 }; // error: cannot deduce element type

const auto &i = expr;
```

The type of `i` is the deduced type of the parameter `u` in the call `f(expr)` of the following invented function template:

```
template <class U> void f(const U& u);
```

—*end example*]

If the ~~list of declarators~~ ***init-declarator-list*** contains more than one ~~declarator~~ ***init-declarator***, the type of each declared variable is determined as described above. If the type ~~deduced for the template parameter `U`~~ **that replaces the occurrence of** `auto` is not the same in each deduction, the program is ill-formed.

[*Example:*

~~`const auto &i = expr;`~~

~~The type of `i` is the deduced type of the parameter `u` in the call `f(expr)` of the following invented function template:~~

~~`template <class U> void f(const U& u);`~~

```
auto x = 5, *y = &x;        // OK: auto is int
auto a = 5, b = { 1, 2 };   // error: different types for auto
```

—*end example*]

1588. Deducing cv-qualified `auto`

Section: 10.1.7.4 [dcl.spec.auto] **Status:** CD3 **Submitter:** Jens Maurer **Date:** 2012-11-18

In an example like

```
const auto x = 3;
```

the intent, clearly, is to make `const int` the type of `x`. It is not clear, however, that the current wording accomplishes this. Because the deduction is based on that of function calls, and because top-level cv-qualifiers are ignored in such deduction, it appears that 10.1.7.4 [dcl.spec.auto] paragraph 6,

> The type deduced for the variable `d` is then the deduced `A` determined using the rules of template argument deduction from a function call (17.9.2.1 [temp.deduct.call]), where `P` is a function template parameter type and the initializer for `d` is the corresponding argument.

incorrectly gives `x` the type `int`.

Proposed resolution (April, 2013):

This issue is resolved by the wording changes in N3638, adopted at the April, 2013 (Bristol) meeting.

977. When is an enumeration type complete?

Section: 10.2 [dcl.enum] **Status:** CD3 **Submitter:** Daniel Krügler **Date:** 3 October, 2009

[Moved to DR at the April, 2013 meeting.]

There is disagreement among implementations as to when an enumeration type is complete. For example,

```
enum E { e = E() };
```

is rejected by some and accepted by another. The Standard does not appear to resolve this question definitively.

See also issue 1482.

Proposed resolution (December, 2012):

This issue is resolved by the resolution of issue 1482.

1439. Lookup and friend template declarations

Section: 10.3.1.2 [namespace.memdef] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2012-01-04

[Moved to DR at the October, 2012 meeting.]

According to 10.3.1.2 [namespace.memdef] paragraph 3,

> If a `friend` declaration in a non-local class first declares a class or function[95] the friend class or function is a member of the innermost enclosing namespace. The name of the friend is not found by unqualified lookup (6.4.1 [basic.lookup.unqual]) or by qualified lookup (6.4.3 [basic.lookup.qual]) until a matching declaration is provided in that namespace scope (either before or after the class definition granting friendship).

This wording does not, but probably should, apply to friend declarations of function templates and class templates as well.

Proposed resolution (February, 2012):

1. Change 6.4.2 [basic.lookup.argdep] paragraph 1 as follows:

   > ...in those namespaces, namespace-scope friend function **or function template** declarations (11.3) not otherwise visible may be found...

2. Change 10.3.1.2 [namespace.memdef] paragraph 3 as follows:

   > Every name first declared in a namespace is a member of that namespace. If a `friend` declaration in a non-local class first declares a class**,** ~~or~~ function**, class template, or function template**[95] the friend ~~class or function~~ is a member of the innermost enclosing namespace. The name of the friend is not found by unqualified lookup (6.4.1 [basic.lookup.unqual]) or by qualified lookup (6.4.3 [basic.lookup.qual]) until a matching declaration is provided in that namespace scope (either before or after the class definition granting friendship). If a friend function **or function template** is called, its name may be found by the name lookup that considers functions from namespaces and classes associated with the types of the function arguments (6.4.2 [basic.lookup.argdep]). If the name in a `friend` declaration...

1477. Definition of a `friend` outside its namespace

Section: 10.3.1.2 [namespace.memdef] **Status:** CD3 **Submitter:** John Spicer **Date:** 2012-03-09

[Moved to DR at the April, 2013 meeting.]

According to 10.3.1.2 [namespace.memdef] paragraph 3,

> Every name first declared in a namespace is a member of that namespace. If a `friend` declaration in a non-local class first declares a class or function[95] the friend class or function is a member of the innermost enclosing namespace. The name of the friend is not found by unqualified lookup (6.4.1 [basic.lookup.unqual]) or by qualified lookup (6.4.3 [basic.lookup.qual]) until a matching declaration is provided in that namespace scope (either before or after the class definition granting friendship).

Taken literally, that would mean the following example is ill-formed:

```
namespace N {
  struct A {
    friend int f();
  };
}
int N::f() { return 0; }
int i = N::f();     // ill-formed: N::f not found
```

because the definition of `N::f` appears in global scope rather than in namespace scope.

Proposed resolution (October, 2012):

Change 10.3.1.2 [namespace.memdef] paragraph 3 as follows:

> Every name first declared in a namespace is a member of that namespace. If a `friend` declaration in a non-local class first declares a class or function[95] the friend class or function is a member of the innermost enclosing namespace. ~~The name of the friend is not found by~~ **The** `friend` **declaration does not by itself make the name visible to** unqualified lookup (6.4.1 [basic.lookup.unqual]) or ~~by~~ qualified lookup (6.4.3 [basic.lookup.qual])**. [*Note:* The name of the friend will be visible in its namespace if** ~~until~~ a matching declaration is provided ~~in that~~ **at** namespace scope (either before or after the class definition granting friendship). **—*end note*]** If a friend function is called...

---

565. Conflict rules for *using-declaration*s naming function templates

Section: 10.3.3 [namespace.udecl] **Status:** CD3 **Submitter:** Paolo Carlini **Date:** 9 March 2006

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

The Standard does not appear to specify what happens for code like the following:

```
namespace one {
  template<typename T> void fun(T);
}

using one::fun;

template<typename T> void fun(T);
```

10.3.3 [namespace.udecl] paragraph 13 does not appear to apply because it deals only with functions, not function templates:

> If a function declaration in namespace scope or block scope has the same name and the same parameter types as a function introduced by a *using-declaration*, and the declarations do not declare the same function, the program is ill-formed.

John Spicer: For function templates I believe the rule should be that if they have the same function type (parameter types and return type) and have identical template parameter lists, the program is ill-formed.

Proposed resolution (August, 2011):

Change 10.3.3 [namespace.udecl] paragraph 14 as follows:

> If a function declaration in namespace scope or block scope has the same name and the same ~~parameter types~~ **parameter-type-list (11.3.5 [dcl.fct])** as a function introduced by a *using-declaration*, and the declarations do not declare the same function, the program is ill-formed. **If a function template declaration in namespace scope has the same name, parameter-type-list, return type, and template parameter list as a function template introduced by a *using-declaration*, the program is ill-formed.** [*Note:* Two *using-declaration*s may introduce functions with the same name and the same ~~parameter types~~ **parameter-type-list**. If, for a call to an unqualified function name, function overload resolution selects the functions introduced by such *using-declaration*s, the function call is ill-formed. [*Example*:...

1475. Errors in `[[carries_dependency]]` example

Section: 10.6.3 [dcl.attr.depend] **Status:** CD3 **Submitter:** Stephan Lavavej **Date:** 2012-03-06

[Moved to DR at the April, 2013 meeting.]

The example in 10.6.3 [dcl.attr.depend] paragraph 4 reads,

```
/* Translation unit A. */

struct foo { int* a; int* b; };
std::atomic<struct foo *> foo_head[10];
int foo_array[10][10];

[[carries_dependency]] struct foo* f(int i) {
  return foo_head[i].load(memory_order_consume);
}

[[carries_dependency]] int g(int* x, int* y) {
  return kill_dependency(foo_array[*x][*y]);
}

/* Translation unit B. */

[[carries_dependency]] struct foo* f(int i);
[[carries_dependency]] int* g(int* x, int* y);

int c = 3;

void h(int i) {
```

```
  struct foo* p;

  p = f(i);
  do_something_with(g(&c, p->a));
  do_something_with(g(p->a, &c));
}
```

There appear to be two errors in this example. First, `g` is declared as returning `int` in TU A but `int*` in TU B. Second, paragraph 6 says,

> Function `g`'s second argument has a `carries_dependency` attribute

but that is not reflected in the example.

Proposed resolution (August, 2012):

1. Change the example in 10.6.3 [dcl.attr.depend] paragraph 4 as follows:

```
/* Translation unit A. */

struct foo { int* a; int* b; };
std::atomic<struct foo *> foo_head[10];
int foo_array[10][10];

[[carries_dependency]] struct foo* f(int i) {
  return foo_head[i].load(memory_order_consume);
}

~~[[carries_dependency]]~~ int g(int* x, int* y [[carries_dependency]]) {
  return kill_dependency(foo_array[*x][*y]);
}

/* Translation unit B. */

[[carries_dependency]] struct foo* f(int i);
~~[[carries_dependency]] int*~~ int g(int* x, int* y [[carries_dependency]]);

int c = 3;

void h(int i) {
  struct foo* p;
  p = f(i);
  do_something_with(g(&c, p->a));
  do_something_with(g(p->a, &c));
}
```

2. Change 10.6.3 [dcl.attr.depend] paragraph 6 as follows:

> Function `g`'s second ~~argument~~ **parameter** has a `carries_dependency` attribute, but its first ~~argument~~ **parameter** does not. Therefore, function `h`'s first call to `g` carries a dependency into `g`, but its second call does not. The implementation might need to insert a fence prior to the second call to `g`.

1297. Misplaced function *attribute-specifier*

Section: 11 [dcl.decl] **Status:** CD3 **Submitter:** Nikolay Ivchenkov **Date:** 2011-04-14

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

There is a contradiction between the grammar of 11 [dcl.decl] paragraph 4 and that of 11.3.5 [dcl.fct] paragraphs 1 and 2 and 11.4.1 [dcl.fct.def.general] paragraph 2 regarding the placement of the optional *exception-specification*: in the former, it immediately follows the *parameter-declaration-clause*, while in the latter it follows the *exception-specification*.

Proposed resolution (August, 2011):

1. Change the grammar in 11 [dcl.decl] paragraph 4 as follows:

> *parameters-and-qualifiers:*
> (*parameter-declaration-clause*) ~~*attribute-specifier-seq*$_{opt}$~~ *cv-qualifier-seq*$_{opt}$
> *ref-qualifier*$_{opt}$ *exception-specification*$_{opt}$ ***attribute-specifier-seq*$_{opt}$**

2. Change the grammar snippet in 16.3.1.1.2 [over.call.object] paragraph 2 as follows:

> `operator` *conversion-type-id* ( ) ~~*attribute-specifier-seq*$_{opt}$~~ *cv-qualifier* ***attribute-specifier-seq*$_{opt}$** ;

1382. Dead code for constructor names

Section: 11 [dcl.decl] **Status:** CD3 **Submitter:** Johannes Schaub **Date:** 2011-08-27

[Moved to DR at the October, 2012 meeting.]

Issue 147 changed the name lookup rules so that a lookup that would have found the injected-class-name of a class will refer to the constructor. However, there still appear to be vestiges of the earlier specification that were not removed by the resolution. For example, the grammar in 11 [dcl.decl] paragraph 4 contains,

> *declarator-id:*
> ... *opt* *id-expression*
> *nested-name-specifier* *opt* *class-name*

It would seem that there is no longer any need for the second line, since a lookup for a *declarator-id* will not produce a *class-name*. Similarly, _N4567_.5.1.1 [expr.prim.general] paragraph 8 still contains the sentence,

> Where *class-name* :: *class-name* is used, and the two *class-name*s refer to the same class, this notation names the constructor (15.1 [class.ctor]).

Proposed resolution (February, 2012):

1. Change 11 [dcl.decl] paragraph 4 as follows:

> ...
> *declarator-id:*
> ... *opt* *id-expression*
> ~~*nested-name-specifier* *opt* *class-name*~~
>
> ~~A *class-name* has special meaning in a declaration of the class of that name and when qualified by that name using the scope resolution operator :: (, 15.1 [class.ctor], 15.4 [class.dtor]).~~

2. Change _N4567_.5.1.1 [expr.prim.general] paragraph 8 as follows:

> ...[*Note:* a class member can be referred to using a *qualified-id* at any point in its potential scope (6.3.7 [basic.scope.class]). —*end note*] ~~Where *class-name* :: *class-name* is used, and the two *class-name*s refer to the same class, this notation names the constructor (15.1 [class.ctor]).~~ Where *class-name* ::~ *class-name* is used...

1528. Repeated *cv-qualifier*s in declarators

Section: 11 [dcl.decl] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2012-07-23

[Moved to DR at the April, 2013 meeting.]

There is no restriction against repeated *cv-qualifier*s in declarators, although there is (in 10.1.7 [dcl.type] paragraph 2) for those appearing in a *decl-specifier-seq*. However, most or all current implementations reject such code. Is a change needed?

Proposed resolution (October, 2012):

Change 10.1.7.1 [dcl.type.cv] paragraph 1 as follows:

> There are two *cv-qualifier*s, `const` and `volatile`. **Each *cv-qualifier* shall appear at most once in a *cv-qualifier-seq*.** If a *cv-qualifier* appears in a *decl-specifier-seq*, the *init-declarator-list* of the declaration shall not be empty. [*Note:*...

---

482. Qualified declarators in redeclarations

Section: 11.3 [dcl.meaning] **Status:** CD3 **Submitter:** Daveed Vandevoorde **Date:** 03 Nov 2004

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

According to 11.3 [dcl.meaning] paragraph 1,

> A *declarator-id* shall not be qualified except for the definition of a member function (12.2.1 [class.mfct]) or static data member (12.2.3 [class.static]) outside of its class, the definition or explicit instantiation of a function or variable member of a namespace outside of its namespace, or 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 (14.3 [class.friend]). When the *declarator-id* is qualified, the declaration shall refer to a previously declared member of the class or namespace to which the qualifier refers...

This restriction prohibits examples like the following:

```
void f();
void ::f();        // error: qualified declarator

namespace N {
  void f();
  void N::f() { }  // error: qualified declarator
}
```

There doesn't seem to be any good reason for disallowing such declarations, and a number of implementations accept them in spite of the Standard's prohibition. Should the Standard be changed to allow them?

Notes from the April, 2006 meeting:

In discussing issue 548, the CWG agreed that the prohibition of qualified declarators inside their namespace should be removed.

Proposed resolution (October, 2006):

Remove the indicated words from 11.3 [dcl.meaning] paragraph 1:

> ...An *unqualified-id* occurring in a *declarator-id* shall be a simple *identifier* except for the declaration of some special functions (15.3 [class.conv], 15.4 [class.dtor], 16.5 [over.oper]) and for the declaration of template specializations or partial specializations (). ~~A *declarator-id* shall not be qualified except for the definition of a member function (12.2.1 [class.mfct]) or static data member (12.2.3 [class.static]) outside of its class, the definition or explicit instantiation of a function or variable member of a namespace outside of its namespace, or 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 (14.3 [class.friend]).~~ When the *declarator-id* is qualified, the declaration shall refer to a previously declared member of the class or namespace to which the qualifier refers, and the member shall not have been introduced by a *using-declaration* in the scope of the class or namespace nominated by the *nested-name-specifier* of the *declarator-id*...

[*Drafting note:* The omission of "outside of its class" here does not give permission for redeclaration of class members; that is still prohibited by 12.2 [class.mem] paragraph 1. The removal of the enumeration of the kinds of declarations in which a *qualified-id* can appear does allow a `typedef` declaration to use a *qualified-id*, which was not permitted before; if that is undesirable, the prohibition can be reinstated here.]

1435. *template-id* as the declarator for a class template constructor

Section: 11.3 [dcl.meaning] **Status:** CD3 **Submitter:** Johannes Schaub **Date:** 2011-12-24

[Moved to DR at the April, 2013 meeting.]

The status of a declaration like the following is unclear:

```
template<typename T> struct A {
  A<T>();
};
```

11.3 [dcl.meaning] paragraph 1 appears to say that it is not allowed, but it is not clear.

Proposed resolution (October, 2012):

1. Change 11.3 [dcl.meaning] paragraph 1 as follows:

   > A list of declarators appears after an optional (Clause 10 [dcl.dcl]) *decl-specifier-seq* (10.1 [dcl.spec]). Each declarator contains exactly one *declarator-id*; it names the identifier that is declared. An *unqualified-id* occurring in a *declarator-id* shall be a simple *identifier* except for the declaration of some special functions (**15.1 [class.ctor],** 15.3 [class.conv], 15.4 [class.dtor], 16.5 [over.oper]) and for the declaration of template specializations or partial specializations (17.8 [temp.spec]). A *declarator-id* shall not...

2. Change 15.1 [class.ctor] paragraph 1 as follows:

   > Constructors do not have names. ~~A special declarator syntax is used to declare or define the constructor. The syntax uses:~~
   >
   > - ~~an optional *decl-specifier-seq* in which each *decl-specifier* is either a *function-specifier* or `constexpr`;~~
   > - ~~the constructor's class name, and~~
   > - ~~a parameter list~~
   >
   > ~~in that order. In such a declaration, optional parentheses around the constructor class name are ignored.~~ **A declaration of a constructor uses a function declarator (11.3.5 [dcl.fct]) of the form**
   >
   > ***ptr-declarator*（*parameter-declaration-clause*）*exception-specification*$_{opt}$ *attribute-specifier-seq*$_{opt}$**
   >
   > **where the *ptr-declarator* consists solely of an *id-expression*, an optional *attribute-specifier-seq*, and optional surrounding parentheses, and the *id-expression* has one of the following forms:**
   >
   > - **in a *member-declaration* that belongs to the *member-specification* of a class but is not a friend declaration (14.3 [class.friend]), the *id-expression* is the injected-class-name (Clause 12 [class]) of the immediately-enclosing class;**
   > - **in a *member-declaration* that belongs to the *member-specification* of a class template but is not a friend declaration, the *id-expression* is a *class-name* that names the current instantiation (17.7.2.1 [temp.dep.type]) of the immediately-enclosing class template; or**
   > - **in a declaration at namespace scope or in a friend declaration, the *id-expression* is a *qualified-id* that names a constructor (6.4.3.1 [class.qual]).**

The *class-name* shall not be a *typedef-name*. In a constructor declaration, each *decl-specifier* in the optional *decl-specifier-seq* shall be `friend`**,** `inline`**,** `explicit`**, or** `constexpr`**.** [*Example*:...

3. Delete 15.1 [class.ctor] paragraph 3:

   ~~A *typedef-name* shall not be used as the *class-name* in the *declarator-id* for a constructor declaration.~~

4. Change 15.1 [class.ctor] paragraph 4 as follows:

   ~~A constructor shall not be~~ ~~`virtual`~~ ~~(13.3 [class.virtual]) or~~ ~~`static`~~ ~~(12.2.3 [class.static]).~~ A constructor can be invoked for a `const`, `volatile` or `const volatile` object. ~~A constructor shall not be declared~~ ~~`const`~~~~,~~ ~~`volatile`~~~~,~~ ~~or~~ ~~`const volatile`~~ ~~(12.2.2.1 [class.this]).~~ `const` and `volatile` semantics (10.1.7.1 [dcl.type.cv]) are not applied on an object under construction. They come into effect when the constructor for the most derived object (4.5 [intro.object]) ends. ~~A constructor shall not be declared with a *ref-qualifier*.~~

5. Change 15.1 [class.ctor] paragraph 9 as follows:

   ~~No return type (not even~~ ~~`void`~~~~) shall be specified for a constructor.~~ A `return` statement in the body of a constructor shall not specify a return value. The address of a constructor shall not be taken.

6. Change 15.4 [class.dtor] paragraphs 1-2 as follows:

   ~~A special declarator syntax using an optional *function-specifier* (10.1.2 [dcl.fct.spec]) followed by ~ followed by the destructor's class name followed by an empty parameter list is used to declare the destructor in a class definition. In such a declaration, the ~ followed by the destructor's class name can be enclosed in optional parentheses; such parentheses are ignored. A *typedef-name* shall not be used as the class-name following the ~ in the declarator for a destructor declaration.~~ **A declaration of a destructor uses a function declarator (11.3.5 [dcl.fct]) of the form**

   ptr-declarator (***parameter-declaration-clause***) ***exception-specification*$_{opt}$ *attribute-specifier-seq*$_{opt}$**

   where the *ptr-declarator* consists solely of an *id-expression*, an optional *attribute-specifier-seq*, and optional surrounding parentheses, and the *id-expression* has one of the following forms:

   - **in a *member-declaration* that belongs to the *member-specification* of a class but is not a friend declaration (14.3 [class.friend]), the *id-expression* is ~ *class-name* and the *class-name* is the injected-class-name (Clause 12 [class]) of the immediately-enclosing class;**
   - **in a *member-declaration* that belongs to the *member-specification* of a class template but is not a friend declaration, the *id-expression* is ~ *class-name* and the *class-name* names the current instantiation (17.7.2.1 [temp.dep.type]) of the immediately-enclosing class template; or**
   - **in a declaration at namespace scope or in a friend declaration, the *id-expression* is *nested-name-specifier* ~ *class-name* and the *class-name* names the same class as the *nested-name-specifier*.**

   The *class-name* shall not be a *typedef-name*. A destructor shall take no arguments (11.3.5 [dcl.fct]). In a destructor declaration, each *decl-specifier* of the optional *decl-specifier-seq* shall be `friend`**,** `inline`**, or** `virtual`**.**

   A destructor is used to destroy objects of its class type. ~~A destructor takes no parameters, and no return type can be specified for it (not even~~ ~~`void`~~~~).~~ The address of a destructor shall not be taken. ~~A destructor shall not be~~ ~~`static`~~~~.~~ A destructor can be invoked for a `const`, `volatile` or `const volatile` object. ~~A destructor shall not be declared~~ ~~`const`~~~~,~~ ~~`volatile`~~ ~~or~~ ~~`const volatile`~~ ~~(12.2.2.1 [class.this]).~~ `const` and `volatile` semantics (10.1.7.1 [dcl.type.cv]) are not applied on an object under destruction. They stop being in effect when the destructor for the most derived object (4.5 [intro.object]) starts. ~~A destructor shall not be declared with a *ref-qualifier*.~~

This resolution also resolves issue 344.

1510. cv-qualified references via `decltype`

Section: 11.3.2 [dcl.ref] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2012-06-14

[Moved to DR at the April, 2013 meeting.]

According to 11.3.2 [dcl.ref] paragraph 1,

> Cv-qualified references are ill-formed except when the cv-qualifiers are introduced through the use of a typedef (10.1.3 [dcl.typedef]) or of a template type argument (17.3 [temp.arg]), in which case the cv-qualifiers are ignored.

There does not appear to be a good reason not to extend this to apply to apply to `decltype`, as well.

Proposed resolution (October, 2012):

Change 11.3.2 [dcl.ref] paragraph 1 as follows:

> ...Cv-qualified references are ill-formed except when the cv-qualifiers are introduced through the use of a ~~typedef~~ ***typedef-name*** (10.1.3 [dcl.typedef]**, 17.1 [temp.param]**) or ~~of a template type argument (17.3 [temp.arg])~~ ***decltype-specifier*** **(10.1.7.2 [dcl.type.simple])**, in which case the cv-qualifiers are ignored. [*Example*:...

---

332. cv-qualified `void` parameter types

Section: 11.3.5 [dcl.fct] **Status:** CD3 **Submitter:** Michiel Salters **Date:** 9 Jan 2002

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

11.3.5 [dcl.fct]/2 restricts the use of void as parameter type, but does not mention CV qualified versions. Since `void f(volatile void)` isn't a callable function anyway, 11.3.5 [dcl.fct] should also ban cv-qualified versions. (BTW, this follows C)

Suggested resolution:

A possible resolution would be to add (cv-qualified) before void in

> The parameter list `(void)` is equivalent to the empty parameter list. Except for this special case, **(cv-qualified)** `void` shall not be a parameter type (though types derived from `void`, such as `void*`, can).

Proposed resolution (August, 2011):

This issue is resolved by the resolution of issue 577.

---

577. `void` in an empty parameter list

Section: 11.3.5 [dcl.fct] **Status:** CD3 **Submitter:** Ben Hutchings **Date:** 22 April 2006

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

11.3.5 [dcl.fct] paragraph 2 says,

> The parameter list `(void)` is equivalent to the empty parameter list.

This special case is intended for C compatibility, but C99 describes it differently (6.7.5.3 paragraph 10):

> The special case of an unnamed parameter of type `void` as the only item in the list specifies that the function has no parameters.

The C99 formulation allows typedefs for `void`, while C++ (and C90) accept only the keyword itself in this role. Should the C99 approach be adopted?

Notes from the October, 2006 meeting:

The CWG did not take a formal position on this issue; however, there was some concern expressed over the treatment of function templates and member functions of class templates if the C++ rule were changed: for a template parameter `T`, would a function taking a single parameter of type `T` become a no-parameter function if it were instantiated with `T = void`?

Proposed resolution (August, 2011):

Change 11.3.5 [dcl.fct] paragraph 4 as follows:

> ...If the *parameter-declaration-clause* is empty, the function takes no arguments. ~~The parameter list `(void)`~~ **A parameter list consisting of a single unnamed parameter of non-dependent type** `void` is equivalent to ~~the~~ **an** empty parameter list. Except for this special case, **a parameter shall not have type *cv*** `void` ~~shall not be a parameter type (though types derived from `void`, such as `void*`, can)~~. If the *parameter-declaration-clause* terminates...

This resolution also resolves issue 332.

1380. Type definitions in *template-parameter parameter-declaration*s

Section: 11.3.5 [dcl.fct] **Status:** CD3 **Submitter:** Daniel Krügler **Date:** 2011-08-22

[Moved to DR at the October, 2012 meeting.]

Although 11.3.5 [dcl.fct] paragraph 9 forbids defining a type in a parameter declaration, and a template parameter declaration is syntactically a *parameter-declaration*, the context in 11.3.5 [dcl.fct] function declarators. It's therefore not completely clear that that prohibition applies to template parameter declarations as well. This should be clarified.

Proposed resolution (February, 2012):

Change 17.1 [temp.param] paragraph 2 as follows:

> ...A storage class shall not be specified in a *template-parameter* declaration. **Types shall not be defined in a *template-parameter* declaration.** [*Note:*...

1394. Incomplete types as parameters of deleted functions

Section: 11.3.5 [dcl.fct] **Status:** CD3 **Submitter:** Johannes Schaub **Date:** 2011-09-11

[Moved to DR at the October, 2012 meeting.]

Currently, 11.3.5 [dcl.fct] paragraph 9 requires that

> The type of a parameter or the return type for a function definition shall not be an incomplete class type (possibly cv-qualified) unless the function definition is nested within the member-specification for that class (including definitions in nested classes defined within the class).

There is no reason for this requirement for a function with a deleted definition, and it would be useful to relax this prohibition in such cases.

Proposed resolution (February, 2012):

Change 11.3.5 [dcl.fct] paragraph 9 as follows:

> Types shall not be defined in return or parameter types. The type of a parameter or the return type for a function definition shall not be an incomplete class type (possibly cv-qualified) unless the function **is deleted (11.4.3 [dcl.fct.def.delete]) or the** definition is nested within the *member-specification* for that class (including definitions in nested classes defined within the class).

1226. Converting a *braced-init-list* default argument

Section: 11.3.6 [dcl.fct.default] **Status:** CD3 **Submitter:** Mike Miller **Date:** 2010-11-19

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

According to the new wording of 11.3.6 [dcl.fct.default] paragraph 5,

> A default argument is implicitly converted (Clause 7 [conv]) to the parameter type.

This is incorrect when the default argument is a *braced-init-list*. That sentence doesn't seem to be necessary, but if it is kept, it should be recast in terms of initialization rather than conversion.

Proposed resolution (August, 2011):

Delete the first sentence of 11.3.6 [dcl.fct.default] paragraph 5:

> ~~A default argument is implicitly converted (Clause 7 [conv]) to the parameter type.~~

1327. *virt-specifier* in a defaulted definition

Section: 11.4.2 [dcl.fct.def.default] **Status:** CD3 **Submitter:** Ryou Ezoe **Date:** 2011-05-29

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

The grammar for defaulted and deleted functions in 11.4.2 [dcl.fct.def.default] and 11.4.3 [dcl.fct.def.delete] does not provide for *virt-specifier*s. Is there a reason for this omission, or was it inadvertent?

Proposed resolution (August, 2011):

1. Change 11.4.2 [dcl.fct.def.default] paragraph 1 as follows:

   > A function definition of the form:
   >
   > *attribute-specifier-seq*$_{opt}$ *decl-specifier-seq*$_{opt}$ *declarator* ***virt-specifier-seq*$_{opt}$** `= default ;`
   >
   > is called an *explicitly-defaulted* definition...

2. Change 11.4.3 [dcl.fct.def.delete] paragraph 1 as follows:

   > A function definition of the form:
   >
   > *attribute-specifier-seq*$_{opt}$ *decl-specifier-seq*$_{opt}$ *declarator* ***virt-specifier-seq*$_{opt}$** `= delete ;`
   >
   > is called a *deleted definition*...

1333. Omission of `const` in a defaulted copy constructor

Section: 11.4.2 [dcl.fct.def.default] **Status:** CD3 **Submitter:** Daniel Krügler **Date:** 2011-06-21

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

Paragraph 1 of 11.4.2 [dcl.fct.def.default] allows an explicitly-defaulted copy constructor or copy assignment operator to have a parameter type that is a reference to non-const, even if the corresponding implicitly-declared function would have a reference to const. However, paragraph 2 says that a copy constructor or copy assignment operator that is defaulted on its first declaration, the parameter type must be exactly the same. Is there a good reason for the stricter rule for a function that is defaulted on its first declaration?

Proposed resolution (August, 2011):

1. Change 11.4.2 [dcl.fct.def.default] paragraph 2 as follows:

   ...If a function is explicitly defaulted on its first declaration,

   - ...
   - ~~in the case of a copy constructor, move constructor, copy assignment operator, or move assignment operator, it shall have the same parameter type as if it had been implicitly declared.~~

2. Change 15.8 [class.copy] paragraph 12 as follows:

   A copy/move constructor for class `X` is trivial if it is not user-provided**, its declared parameter type is the same as if it had been implicitly declared,** and if...

3. Change 15.8 [class.copy] paragraph 25 as follows:

   A copy/move assignment operator for class X is trivial if it is not user-provided**, its declared parameter type is the same as if it had been implicitly declared,** and if...

1355. Aggregates and "user-provided" constructors

Section: 11.4.2 [dcl.fct.def.default] **Status:** CD3 **Submitter:** Sean Hunt **Date:** 2011-08-16

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

The definition of "user-provided" given in 11.4.2 [dcl.fct.def.default] paragraph 4 applies only to special member functions, while the definition of an aggregate in 11.6.1 [dcl.init.aggr] paragraph 1 relies on that term in identifying constructors that make a class a non-aggregate. As a result, a class with a non-special constructor is considered an aggregate.

Proposed resolution (August, 2011):

Change 11.4.2 [dcl.fct.def.default] paragraph 4 as follows:

A ~~special member~~ function is user-provided if it is user-declared and not explicitly defaulted or deleted on its first declaration...

[Drafting note: This makes a class with only a deleted initializer-list constructor an aggregate.]

1093. Value-initializing non-objects

Section: 11.6 [dcl.init] **Status:** CD3 **Submitter:** Daniel Krügler **Date:** 2010-07-17

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

11.6 [dcl.init] paragraph 7 only describes how to initialize objects:

To *value-initialize* an object of type `T` means:

However, 8.2.3 [expr.type.conv] paragraph 2 calls for value-initializing prvalues, which in the case of scalar types are not objects:

The expression `T()`, where `T` is a *simple-type-specifier* or *typename-specifier* for a non-array complete object type or the (possibly cv-qualified) `void` type, creates a prvalue of the specified type, which is value-initialized (11.6 [dcl.init]; no initialization is done for the `void()` case).

Proposed resolution (August, 2011):

Change 8.2.3 [expr.type.conv] paragraph 2 as follows:

The expression `T()`, where `T` is a *simple-type-specifier* or *typename-specifier* for a non-array complete object type or the (possibly cv-qualified) `void` type, creates a prvalue of the specified ~~type,which is value-initialized (11.6 [dcl.init]~~ **type, whose value is that produced by value-initializing (11.6 [dcl.init]) an object of type** `T`; no initialization is done for the `void()` case~~)~~. [*Note:*...

1301. Value initialization of union

Section: 11.6 [dcl.init] **Status:** CD3 **Submitter:** Jason Merrill **Date:** 2011-04-18

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

According to 11.6 [dcl.init] paragraph 7,

> To *value-initialize* an object of type `T` means:
>
> - if `T` is a (possibly cv-qualified) class type (Clause 12 [class]) with a user-provided constructor (15.1 [class.ctor]), then the default constructor for `T` is called (and the initialization is ill-formed if `T` has no accessible default constructor);
> - if `T` is a (possibly cv-qualified) non-union class type without a user-provided constructor, then the object is zero-initialized and, if `T`'s implicitly-declared default constructor is non-trivial, that constructor is called.
> - if `T` is an array type, then each element is value-initialized;
> - otherwise, the object is zero-initialized.

This suggests that for

```
struct A { A() = delete; };
union B { A a };
int main()
{
  B();
}
```

a `B` temporary is created and zero-initialized, even though its default constructor is deleted. We should strike "non-union" and also the "if...non-trivial" condition, since we can have a trivial deleted constructor.

Proposed resolution (August, 2011):

1. Change 11.6 [dcl.init] paragraph 7 as follows:

   > To *value-initialize* an object of type `T` means:
   >
   > - if `T` is a (possibly cv-qualified) class type (Clause 12 [class]) with **either no default constructor (15.1 [class.ctor]) or** a **default constructor that is** user-provided **or deleted** ~~constructor (15.1 [class.ctor])~~, then the **object is default-initialized** ~~default constructor for `T` is called (and the initialization is ill-formed if `T` has no accessible default constructor)~~;
> - if `T` is a (possibly cv-qualified) non-union class type without a user-provided **or deleted default** constructor, then the object is zero-initialized and, if ~~`T`'s implicitly-declared default constructor is~~ `T` **has a** non-trivial **default constructor**, ~~that constructor is called.~~ **default-initialized;**
> - ...

2. Change 11.6.4 [dcl.init.list] paragraph 3 as follows:

   > List-initialization of an object or reference of type `T` is defined as follows:
   >
   > - ~~If the initializer list has no elements and `T` is a class type with a default constructor, the object is value-initialized.~~
> - ~~Otherwise, if~~ **If** `T` is an aggregate, aggregate initialization is performed (11.6.1 [dcl.init.aggr]). [*Example:*...
> - **Otherwise, if the initializer list has no elements and `T` is a class type with a default constructor, the object is value-initialized.**
> - ...

This resolution also resolves issues 1324 and 1368.

1324. Value initialization and defaulted constructors

Section: 11.6 [dcl.init] **Status:** CD3 **Submitter:** Daniel Krügler **Date:** 2011-05-22

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

One would expect that an example like

```
struct B {
 B(const B&) = default;
};

B b{};
```

would invoke value-initialization, but (because it does not have a default constructor), the logic ladder of 11.6.4 [dcl.init.list] paragraph 3 makes it aggregate initialization instead:

- If the initializer list has no elements and `T` is a class type with a default constructor, the object is value-initialized.
- Otherwise, if `T` is an aggregate, aggregate initialization is performed (11.6.1 [dcl.init.aggr]).

Proposed resolution (August, 2011):

This issue is resolved by the resolution of issue 1301.

1368. Value initialization and defaulted constructors (part 2)

Section: 11.6 [dcl.init] **Status:** CD3 **Submitter:** Jason Merrill **Date:** 2011-06-28

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

According to the current rules of 11.6 [dcl.init], given a class like

```
struct A {
  int i;
  A() = default;
  A(int i): i(i) { }
};
```

value-initialization leaves `A::i` uninitialized. This seems like an oversight.

Proposed resolution (August, 2011):

This issue is resolved by the resolution of issue 1301.

1502. Value initialization of unions with member initializers

Section: 11.6 [dcl.init] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2012-05-06

[Moved to DR at the April, 2013 meeting.]

According to 11.6 [dcl.init] paragraph 7,

> To *value-initialize* an object of type `T` means:
>
> - if `T` is a (possibly cv-qualified) class type (Clause 12 [class]) with either no default constructor (15.1 [class.ctor]) or a default constructor that is user-provided or deleted, then the object is default-initialized;
> - if `T` is a (possibly cv-qualified) non-union class type without a user-provided or deleted default constructor, then the object is zero-initialized and, if `T` has a non-trivial default constructor, default-initialized;
> - if `T` is an array type, then each element is value-initialized;
> - otherwise, the object is zero-initialized.

In an example like

```
union U { int a = 5; };
int main() { return U().a; }
```

this means that the value returned is `0`, because none of the first three bullets apply. Should the "non-union" restriction be dropped from the second bullet?

Proposed resolution (October, 2012):

Change 11.6 [dcl.init] paragraph 7 as follows:

> To *value-initialize* an object of type `T` means:
>
> - ...
> - if `T` is a (possibly cv-qualified) ~~non-union~~ class type without a user-provided or deleted default constructor, then the object is zero-initialized and, if `T` has a non-trivial default constructor, default-initialized;

- ...

1507. Value initialization with trivial inaccessible default constructor

Section: 11.6 [dcl.init] **Status:** CD3 **Submitter:** Johannes Schaub **Date:** 2012-06-01

[Moved to DR at the April, 2013 meeting.]

According to 11.6 [dcl.init] paragraph 7, a trivial default constructor is not used in value initialization, so the following example would appear to be well-formed:

```
struct A { private: A() = default; };
int main() { A(); }
```

Proposed resolution (February, 2013):

1. Change 11.6 [dcl.init] paragraph 7 as follows:

   To *default-initialize* an object of type `T` means:

   - if `T` is a (possibly cv-qualified) class type (Clause 12 [class]), the default constructor **(15.1 [class.ctor])** for `T` is called (and the initialization is ill-formed if `T` has no ~~accessible default constructor~~ **default constructor or overload resolution (16.3 [over.match]) results in an ambiguity or in a function that is deleted or inaccessible from the context of the initialization**);

   - ...

2. Change 11.6 [dcl.init] paragraph 8 as follows:

   To *value-initialize* an object of type `T` means:

   - ...

   - if `T` is a (possibly cv-qualified) non-union class type without a user-provided or deleted default constructor, then the object is zero-initialized and **the semantic constraints for default-initialization are checked**, **and** if `T` has a non-trivial default constructor, **the object is** default-initialized;

   - ...

1295. Binding a reference to an rvalue bit-field

Section: 11.6.3 [dcl.init.ref] **Status:** CD3 **Submitter:** Nikolay Ivchenkov **Date:** 2011-04-14

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

Consider the following example:

```
struct X {
  unsigned bitfield : 4;
};
int main() {
  X x = { 1 };
  unsigned const &ref = static_cast<X &&>(x).bitfield;
}
```

According to 11.6.3 [dcl.init.ref] paragraph 5, `ref` is bound to the bit-field xvalue.

Proposed resolution (August, 2011):

Change the indicated sub-bullet of 11.6.3 [dcl.init.ref] paragraph 5 as follows:

- is an xvalue **(but is not a bit-field)**, class prvalue, array prvalue or function lvalue and "*cv1* `T1`" is reference-compatible with "*cv2* `T2`", or

---

1328. Conflict in reference binding vs overload resolution

Section: 11.6.3 [dcl.init.ref] **Status:** CD3 **Submitter:** Johannes Schaub **Date:** 2011-05-31

[Moved to DR at the April, 2013 meeting.]

According to 16.3.1.6 [over.match.ref] paragraph 1, the determination of the candidate functions is based on whether 11.6.3 [dcl.init.ref] requires an lvalue result or an rvalue result. It is not sufficiently clear exactly what this means, particularly with respect to function lvalues

and rvalues.

Proposed resolution (August, 2011):

1. Change 16.3.1.6 [over.match.ref] paragraph 1 as follows:

   - The conversion functions of S and its base classes are considered, except that for copy-initialization, only the non-explicit conversion functions are considered. Those that are not hidden within S and yield type "lvalue reference to *cv2* T2" (when ~~11.6.3 [dcl.init.ref] requires an lvalue result~~ **initializing an lvalue reference or an rvalue reference to function**) or "*cv2* T2" or "rvalue reference to *cv2* T2" (when ~~11.6.3 [dcl.init.ref] requires an rvalue result~~ **initializing an rvalue reference or an lvalue reference to function**), where "*cv1* T" is reference-compatible (11.6.3 [dcl.init.ref]) with "*cv2* T2", are candidate functions.

2. Change 16.3.3 [over.match.best] paragraph 1 as follows:

   - ...
   - the context is an initialization by user-defined conversion... or if not that,
   - **the context is an initialization by conversion function for direct reference binding (16.3.1.6 [over.match.ref]) of a reference to function type, the return type of F1 is the same kind of reference (i.e. lvalue or rvalue) as the reference being initialized, and the return type of F2 is not [*Example:***

     ```
     template <class T> struct A {
       operator T&();  // #1
       operator T&&(); // #2
     };
     typedef int Fn();
     A<Fn> a;
     Fn& lf = a;       // calls #1
     Fn&& rf = a;      // calls #2
     ```

     —*end example*] or, if not that,

   - F1 is a non-template function...

1401. Similar types and reference compatibility

Section: 11.6.3 [dcl.init.ref] **Status:** CD3 **Submitter:** Daniel Krügler **Date:** 2011-10-03

[Moved to DR at the October, 2012 meeting.]

The definition of reference-compatible types in 11.6.3 [dcl.init.ref] paragraph 4 allows the types to differ in top-level cv-qualification, but it does not encompass the deeper added cv-qualification permitted for "similar types" (7.5 [conv.qual]). This seems surprising and could lead to errors resulting from the fact that the reference will be bound to a temporary and not to the original object in the initializer.

Proposed resolution (February, 2012):

1. Change 11.6.3 [dcl.init.ref] paragraph 4 as follows:

   Given types "*cv1* T1" and "*cv2* T2," "*cv1* T1" is reference-related to "*cv2* T2" if T1 is the same type as T2, or T1 is a base class of T2. "*cv1* T1" is reference-compatible with "*cv2* T2" if T1 is reference-related to T2 and *cv1* is the same cv-qualification as, or greater cv-qualification than, *cv2*. ~~For purposes of overload resolution, cases for which *cv1* is greater cv-qualification than *cv2* are identified as *reference-compatible with added qualification* (see 16.3.3.2 [over.ics.rank]).~~ In all cases...

2. Delete 16.3.3.1.4 [over.ics.ref] paragraph 5:

   ~~The binding of a reference to an expression that is *reference-compatible with added qualification* influences the rank of a standard conversion; see 16.3.3.2 [over.ics.rank] and 11.6.3 [dcl.init.ref].~~

[Drafting note: CWG decided not to make a substantive change for this issue, but the investigation discovered that the term defined by these two citations is not actually used and could be removed.]

1270. Brace elision in array temporary initialization

Section: 11.6.4 [dcl.init.list] **Status:** CD3 **Submitter:** Jason Merrill **Date:** 2011-03-23

[Moved to DR at the October, 2012 meeting.]

Issue 1232 extended the language to allow creation of array temporaries using initializer lists. However, such initializer lists must be "completely braced;" the elision of braces described in 11.6.1 [dcl.init.aggr] paragraph 11 applies only

> In a declaration of the form
>
> ```
> T x = { a };
> ```

This restriction prevents plausible uses like

```
array<int, 3> f() {
  return { 1, 2, 3 };
}
```

Proposed resolution (February, 2012):

Change 11.6.1 [dcl.init.aggr] paragraph 11 as follows:

> ~~In a declaration of the form~~
>
> ~~`T x = { a };`~~
>
> ~~braces~~ **Braces** can be elided in an *initializer-list* as follows. ~~[*Footnote:* Braces cannot be elided in other uses of list-initialization. *— end footnote*~~]

1288. Reference list initialization

Section: 11.6.4 [dcl.init.list] **Status:** CD3 **Submitter:** Daniel Krügler **Date:** 2011-04-06

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

One might expect that in an example like

```
int i;
int & ir{i};
```

`ir` would bind directly to `i`. However, according to 11.6.4 [dcl.init.list] paragraph 3, this example creates a temporary of type `int` and binds the reference to that temporary:

- ...
- Otherwise, if `T` is a reference type, a prvalue temporary of the type referenced by `T` is list-initialized, and the reference is bound to that temporary...
- Otherwise, if the initializer list has a single element, the object or reference is initialized from that element...

Also, the "or reference" in the last bullet is dead code, as a reference initialization is always handled by the preceding bullet.

Proposed resolution (August, 2011):

Change 11.6.4 [dcl.init.list] paragraph 3 as follows:

- ...
- Otherwise, if `T` is a class type, constructors are considered...
- ~~Otherwise, if `T` is a reference type, a prvalue temporary of the type referenced by `T` is list-initialized, and the reference is bound to that temporary. [*Note:* As usual, the binding will fail and the program is ill-formed if the reference type is an lvalue reference to a non-const type. *— end note*] [*Example:* ... *— end example*]~~
- Otherwise, if the initializer list has a single element **of type** `E` **and either** `T` **is not a reference type or its referenced type is reference-related to** `E`, the object or reference is initialized from that element; if a narrowing conversion (see below) is required to convert the element to `T`, the program is ill-formed. [*Example:*...
- **Otherwise, if** `T` **is a reference type, a prvalue temporary of the type referenced by** `T` **is list-initialized, and the reference is bound to that temporary. [*Note:* As usual, the binding will fail and the program is ill-formed if the reference type is an lvalue reference to a non-const type. *— end note*] [*Example:* ... *— end example*]**
- Otherwise, if the initializer list has no elements...
- ...

1290. Lifetime of the underlying array of an `initializer_list` member

Section: 11.6.4 [dcl.init.list] **Status:** CD3 **Submitter:** James Dennett **Date:** 2011-04-08

[Moved to DR at the October, 2012 meeting.]

A question has arisen over expected behavior when an `initializer_list` is a non-static data member of a class. Initialization of an `initializer_list` is defined in terms of construction from an implicitly allocated array whose lifetime "is the same as that of the `initializer_list` object". That would mean that the array needs to live as long as the `initializer_list` does, which would on the face of it appear to require the array to be stored in something like a `std::unique_ptr<T[]>` within the same class (if the member is initialized in this manner).

It would be surprising if that was the intent, but it would make `initializer_list` usable in this context.

It would also be reasonable if this behaved similarly to binding temporaries to reference members (i.e., "temporary bound to a reference member in a constructor's *ctor-initializer* (15.6.2 [class.base.init]) persists until the constructor exits."), though this approach would probably prevent use of an `initializer_list` member in that context.

Proposed resolution (February, 2012):

1. Change 11.6.4 [dcl.init.list] paragraphs 5-6 as follows:

> An object of type `std::initializer_list<E>` is constructed from an initializer list as if the implementation allocated ~~an~~ **a temporary** array of *N* elements of type `E`, where...
>
> ~~The lifetime of the array is the same as that of the `initializer_list` object.~~ **The array has the same lifetime as any other temporary object (15.2 [class.temporary]), except that initializing an `initializer_list` object from the array extends the lifetime of the array exactly like binding a reference to a temporary.** [*Example:*
>
> ```
> typedef std::complex<double> cmplx;
> std::vector<cmplx> v1 = { 1, 2, 3 };
>
> void f() {
>   std::vector<cmplx> v2{ 1, 2, 3 };
>   std::initializer_list<int> i3 = { 1, 2, 3 };
> }
>
> struct A {
>   std::initializer_list<int> i4;
>   A(): i4{1,2,3} { }  // creates an A with a dangling reference
> };
> ```
>
> For `v1` and `v2`, the `initializer_list` object **is a parameter in a function call, so the** ~~and~~ array created for `{ 1, 2, 3 }` ~~have~~ **has** full-expression lifetime. For `i3`, the `initializer_list` object **is a variable, so the** ~~and~~ array ~~have automatic~~ **persists for the** lifetime **of the variable. For `i4`, the `initializer_list` object is initialized in a constructor's *ctor-initializer*, so the array persists only until the constructor exits, and so any use of the elements of `i4` after the constructor exits produces undefined behavior**. —*end example*] [*Note:* The implementation is free to allocate the array in read-only memory if an explicit array with the same initializer could be so allocated. —*end note*]

2. Change 15.2 [class.temporary] paragraph 5 as follows:

> The second context is when a reference is bound to a temporary. **[*Footnote:* The same rules apply to initialization of an `initializer_list` object (11.6.4 [dcl.init.list]) with its underlying temporary array. —*end footnote*]** The temporary to which...

1418. Type of `initializer_list` backing array

Section: 11.6.4 [dcl.init.list] **Status:** CD3 **Submitter:** Johannes Schaub **Date:** 2011-11-19

[Moved to DR at the October, 2012 meeting.]

According to 11.6.4 [dcl.init.list] paragraph 5, the elements of the backing array for an object of type `std::initializer_list<E>` are of type `E`. This is contradicted by the wording of 21.9 [support.initlist] paragraph 2.

Proposed resolution (February, 2012):

Change 11.6.4 [dcl.init.list] paragraph 5 as follows:

> An object of type `std::initializer_list<E>` is constructed from an initializer list as if the implementation allocated an array of *N* elements of type `const E`, where *N* is the number of elements in the initializer list. Each element of that array is copy-initialized with the corresponding element of the initializer list, and the `std::initializer_list<E>` object is constructed to refer to that array. If a narrowing conversion is required to initialize any of the elements, the program is ill-formed. [*Example:*
>
> ```
> struct X {
>   X(std::initializer_list<double> v);
> };
> X x{ 1,2,3 };
> ```
>
> The initialization will be implemented in a way roughly equivalent to this:
>
> ```
> const double __a[3] = {double{1}, double{2}, double{3}};
> X x(std::initializer_list<double>(__a, __a+3));
> ```
>
> assuming that the implementation can construct an initializer_list object with a pair of pointers. —*end example*]

1449. Narrowing conversion of negative value to unsigned type

Section: 11.6.4 [dcl.init.list] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2012-01-28

[Moved to DR at the October, 2012 meeting.]

According to 11.6.4 [dcl.init.list] paragraph 7, an implicit conversion

> from an integer type or unscoped enumeration type to an integer type that cannot represent all the values of the original type, except where the source is a constant expression and the actual value after conversion will fit into the target type and will produce the original value when converted back to the original type.

As is made plain in the examples in that paragraph, a conversion of a negative value to an unsigned type is intended to be a narrowing conversion; however, the phrase "actual value after conversion" makes this intent unclear, especially since the round-trip conversion between signed and unsigned types might well yield the original value.

Proposed resolution (February, 2012):

Change 11.6.4 [dcl.init.list] paragraph 7 as follows:

> A *narrowing conversion* is an implicit conversion
>
> - ...
> - from an integer type or unscoped enumeration type to an integer type that cannot represent all the values of the original type, except where the source is a constant expression ~~and the actual value after conversion~~ **whose value after integral promotions** will fit into the target type ~~and will produce the original value when converted back to the original type~~.
>
> [*Note:*...

1494. Temporary initialization for reference binding in list-initialization

Section: 11.6.4 [dcl.init.list] **Status:** CD3 **Submitter:** Steve Adamczyk **Date:** 2012-04-12

[Moved to DR at the April, 2013 meeting.]

One of the bullets in 11.6.4 [dcl.init.list] paragraph 3 reads,

> Otherwise, if T is a reference type, a prvalue temporary of the type referenced by T is list-initialized, and the reference is bound to that temporary.

This does not say whether the initialization of the temporary is direct-list-initialization, copy-list-initialization, or the same kind of list-initialization as the top-level initialization.

Proposed resolution (December, 2012):

Change 11.6.4 [dcl.init.list] paragraph 3 as follows:

- ...
- Otherwise, if `T` is a reference type, a prvalue temporary of the type referenced by `T` is ~~list-initialized~~ **copy-list-initialized or direct-list-initialized, depending on the kind of initialization for the reference**, and the reference is bound to that temporary. [*Note:* As usual...
- ...

1506. Value category of `initializer_list` object

Section: 11.6.4 [dcl.init.list] **Status:** CD3 **Submitter:** Steve Adamczyk **Date:** 2012-05-29

[Moved to DR at the April, 2013 meeting.]

One of the bullets in 11.6.4 [dcl.init.list] paragraph 3 says,

> Otherwise, if `T` is a specialization of `std::initializer_list<E>`, an `initializer_list` object is constructed as described below and used to initialize the object according to the rules for initialization of an object from a class of the same type (11.6 [dcl.init]).

This does not, but should, say whether the `initializer_list` object is treated as an lvalue or prvalue for the purpose of the 11.6 [dcl.init] initialization.

Proposed resolution (October, 2012):

Change 11.6.4 [dcl.init.list] paragraph 3 as follows:

> List-initialization of an object or reference of type `T` is defined as follows:
>
> - ...

- Otherwise, if `T` is a specialization of `std::initializer_list<E>`, ~~an~~ **a prvalue** `initializer_list` object is constructed as described below and used to initialize the object according to the rules for initialization of an object from a class of the same type (11.6 [dcl.init]).
- ...

1522. Access checking for `initializer_list` array initialization

Section: 11.6.4 [dcl.init.list] **Status:** CD3 **Submitter:** John Spicer **Date:** 2012-07-12

[Moved to DR at the April, 2013 meeting.]

In constructing an `initializer_list` object from an initializer list, 11.6.4 [dcl.init.list] paragraph 5 says of the underlying array,

> Each element of that array is copy-initialized with the corresponding element of the initializer list

It would probably be good to mention that the copy/move constructor for this copy must be accessible in the context in which the initialization occurs.

Proposed resolution (October, 2012):

Change 11.6.4 [dcl.init.list] paragraph 4 as follows:

> ...Each element of that array is copy-initialized with the corresponding element of the initializer list, and the `std::initializer_list<E>` object is constructed to refer to that array. **[*Note:* A constructor or conversion function selected for the copy shall be accessible (Clause 14 [class.access]) in the context of the initializer list. —*end note*]** If a narrowing conversion is required...

2150. Initializer list array lifetime

Section: 11.6.4 [dcl.init.list] **Status:** CD3 **Submitter:** Hubert Tong **Date:** 2015-06-26

The resolution of issue 1696 appears to have removed the wording that makes the initialization of `A::i4` in 11.6.4 [dcl.init.list] paragraph 6 create a dangling reference:

> The array has the same lifetime as any other temporary object (15.2 [class.temporary]), except that initializing an initializer_list object from the array extends the lifetime of the array exactly like binding a reference to a temporary. [*Example:*
>
> ```
> typedef std::complex<double> cmplx;
> std::vector<cmplx> v1 = { 1, 2, 3 };
>
> void f() {
>   std::vector<cmplx> v2{ 1, 2, 3 };
>   std::initializer_list<int> i3 = { 1, 2, 3 };
> }
>
> struct A {
>   std::initializer_list<int> i4;
>   A() : i4{ 1, 2, 3 } {} // creates an A with a dangling reference
> };
> ```
>
> For `v1` and `v2`, the `initializer_list` object is a parameter in a function call, so the array created for `{ 1, 2, 3 }` has full-expression lifetime. For `i3`, the `initializer_list` object is a variable, so the array persists for the lifetime of the variable. For `i4`, the `initializer_list` object is initialized in a constructor's ctor-initializer, so the array persists only until the constructor exits, and so any use of the elements of `i4` after the constructor exits produces undefined behavior. —*end example*]

Binding a reference to a temporary in a *mem-initializer* or default member initializer is now ill-formed, per 15.6.2 [class.base.init] paragraphs 8 and 11, which undercuts the description here.

Notes from the October, 2015 meeting:

The example is incorrect and will be fixed editorially. The issue is left in "review" status to check that the editorial change has been made.

1318. Syntactic ambiguities with `final`

Section: 12 [class] **Status:** CD3 **Submitter:** Johannes Schaub **Date:** 2011-05-14

[Moved to DR at the April, 2013 meeting.]

The ambiguity in an example like

```
struct A final { };
```

is resolved by 5.10 [lex.name] paragraph 2 to be the declaration of a variable named `final`:

> any ambiguity as to whether a given *identifier* has a special meaning is resolved to interpret the token as a regular *identifier*.

Similarly, in an example like

```
struct C { constexpr operator int() { return 5; } };
struct A {
 struct B final : C{};
};
```

`final` is taken as the name of a bit-field member rather than as the name of a nested class.

Proposed resolution (October, 2012):

1. Change 5.10 [lex.name] paragraph 2 as follows:

> The identifiers in Table 3 have a special meaning when appearing in a certain context. When referred to in the grammar, these identifiers are used explicitly rather than using the *identifier* grammar production. **Unless otherwise specified,** any ambiguity as to whether a given identifier has a special meaning is resolved to interpret the token as a regular *identifier*.

2. Change 12 [class] paragraph 3 as follows:

> If a class is marked with the *class-virt-specifier* `final` and it appears as a *base-type-specifier* in a *base-clause* (Clause 13 [class.derived]), the program is ill-formed. **Whenever a *class-key* is followed by a *class-head-name*, the *identifier*** `final`**, and a colon or left brace,** `final` **is interpreted as a *class-virt-specifier*. [*Example:***
>
> ```
> struct A;
> struct A final {};      // OK: definition of struct A,
>                         // not value-initialization of variable final
>
> struct X {
>  struct C { constexpr operator int() { return 5; } };
>  struct B final : C{};  // OK: definition of nested class B,
>                         // not declaration of a bit-field member final
> };
> ```
>
> ***— end example***]

1363. Triviality vs multiple default constructors

Section: 12 [class] **Status:** CD3 **Submitter:** Sean Hunt **Date:** 2011-08-16

[Moved to DR at the October, 2012 meeting.]

The requirements for a trivial class include having "a trivial default constructor" (12 [class] paragraph 6). However, with an explicitly-defaulted default constructor and other constructors with default arguments, it is possible to have multiple default constructors. Such a class cannot be default-initialized and thus should probably be considered non-trivial.

Proposed resolution (February, 2012):

Change 12 [class] paragraph 6 as follows:

> ...A *trivial class* is a class that has a ~~trivial~~ default constructor (15.1 [class.ctor])**, has no non-trivial default constructors,** and is trivially copyable...

1411. More on global scope `::` in *nested-name-specifier*

Section: 12 [class] **Status:** CD3 **Submitter:** David Blaikie **Date:** 2011-10-28

[Moved to DR at the April, 2013 meeting.]

The resolution of issue 355 failed to update the grammar for *class-head-name* in 12 [class] paragraph 1 and removed the optional `::` from *class-or-decltype* in 13 [class.derived] paragraph 1, with the result that it is still not possible to globally-qualify a class name in a class definition, and the ability to globally-qualify a base class name has been lost.

Proposed resolution (February, 2012):

Change the grammar in _N4567_.5.1.1 [expr.prim.general] paragraph 8 as follows:

qualified-id:
nested-name-specifier `template`$_{opt}$ *unqualified-id*
~~*:: identifier*~~
~~*:: operator-function-id*~~
~~*:: literal-operator-id*~~
~~*:: template-id*~~

nested-name-specifier:
:: ~~*opt*~~
~~*opt*~~ *type-name* ::
~~*opt*~~ *namespace-name* ::
decltype-specifier ::
nested-name-specifier identifier ::
nested-name-specifier `template`~~*opt*~~ *simple-template-id* ::

1308. Completeness of class type within an *exception-specification*

Section: 12.2 [class.mem] **Status:** CD3 **Submitter:** Nikolay Ivchenkov **Date:** 2011-05-03

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

According to 12.2 [class.mem] paragraph 2,

> A class is considered a completely-defined object type (6.9 [basic.types]) (or complete type) at the closing } of the *class-specifier*. Within the class *member-specification*, the class is regarded as complete within function bodies, default arguments, *exception-specification*s, and *brace-or-equal-initializer*s for non-static data members (including such things in nested classes). Otherwise it is regarded as incomplete within its own class *member-specification*.

With the advent of the `noexcept` operator, treating the class type as complete in *exception-specification*s is obviously not possible, e.g.,

```
struct X {
  // should X be considered as complete here?
  static void create() noexcept(noexcept(X()));
  X() noexcept(!noexcept(X::create()));
};
```

Proposed resolution (August, 2011):

1. Change 12.2 [class.mem] paragraph 2 as follows:

> A class is considered a completely-defined object type (6.9 [basic.types]) (or complete type) at the closing } of the *class-specifier*. Within the class *member-specification*, the class is regarded as complete within function bodies, default arguments, ~~*exception-specifications*~~, and *brace-or-equal-initializers* for non-static data members (including such things in nested classes). Otherwise it is regarded as incomplete within its own class *member-specification*.

2. Change 18.4 [except.spec] paragraph 2 as follows:

> ...A type denoted in an *exception-specification* shall not denote an incomplete type **other than a class currently being defined**. A type denoted in an *exception-specification* shall not denote a pointer or reference to an incomplete type, other than ***cv*** `void`~~`*`~~, ~~`const void*`~~, ~~`volatile void*`~~, or ~~`const volatile void*`~~ **or a pointer or reference to a class currently being defined**. A type *cv* `T`, "array of `T`", or "function returning `T`" denoted in an *exception-specification* is adjusted to type `T`, "pointer to `T`", or "pointer to function returning `T`", respectively.

Note:

This change was subsequently removed by the resolution of issue 1330.

1357. *brace-or-equal-initializer*s for function and typedef members

Section: 12.2 [class.mem] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2011-08-16

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

The grammar allows a *brace-or-equal-initializer* for any class member with a *member-declarator*, including typedef members and member function declarations, and there is no semantic restriction forbidding those forms, either.

Proposed resolution (August, 2011):

In 12.2 [class.mem], delete paragraph 4 and change paragraph 5 as follows:

> ~~A member can be initialized using a constructor; see 15.1 [class.ctor]. [*Note:* See Clause 15 [special] for a description of constructors and other special member functions. — *end note*]~~

> A ~~member can be initialized using a~~ *brace-or-equal-initializer* **shall appear only in the declaration of a data member**. (For static data members, see 12.2.3.2 [class.static.data]; for non-static data members, see 15.6.2 [class.base.init]).

1425. Base-class subobjects of standard-layout structs

Section: 12.2 [class.mem] **Status:** CD3 **Submitter:** Ville Voutilainen **Date:** 2011-12-07

[Moved to DR at the April, 2013 meeting.]

According to 12.2 [class.mem] paragraph 20,

> A pointer to a standard-layout struct object, suitably converted using a `reinterpret_cast`, points to its initial member (or if that member is a bit-field, then to the unit in which it resides) and vice versa.

Given a standard-layout struct in which the non-static data members are in a base class (and hence the derived class is empty), it is not clear what the "initial member" is. Presumably the intent behind allowing such standard-layout classes was to treat the base class object and its first non-static data member as the initial member of the derived class, but this does not appear to be spelled out anywhere.

Proposed resolution (February, 2012):

Change 12.2 [class.mem] paragraph 20 as follows:

> ~~A pointer to a standard-layout struct object, suitably converted using a `reinterpret_cast`, points to its initial member (or if that member is a bit-field, then to the unit in which it resides) and vice versa.~~ **If a standard-layout class object has any non-static data members, its address is the same as the address of its first non-static data member. Otherwise, its address is the same as the address of its first base class subobject (if any).** [*Note:*...

1306. Modifying an object within a `const` member function

Section: 12.2.2.1 [class.this] **Status:** CD3 **Submitter:** James Kanze **Date:** 2011-04-26

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

According to 12.2.2.1 [class.this] paragraph 2,

> In a `const` member function, the object for which the function is called is accessed through a `const` access path; therefore, a `const` member function shall not modify the object and its non-static data members.

This is clearly overstating the case: `mutable` members can be modified, a `const_cast` can be used, and class member access expressions not involving `this` can also allow the object to be modified. The effect of the cv-qualification of a member function on the type of `*this` is clear from the preceding paragraph; this statement appears both unnecessary and incorrect.

Proposed resolution (August, 2011):

Merge 12.2.2.1 [class.this] paragraphs 1 and 2 and change the text as follows:

> In the body of a non-static (12.2.1 [class.mfct]) member function, the keyword `this` is a prvalue expression whose value is the address of the object for which the function is called. The type of `this` in a member function of a class `X` is `X*`. If the member function is declared `const`, the type of `this` is `const X*`, if the member function is declared `volatile`, the type of `this` is `volatile X*`, and if the member function is declared `const volatile`, the type of `this` is `const volatile X*`. ~~In~~ **[*Note:* thus in** a `const` member function, the object for which the function is called is accessed through a `const` access path~~; therefore, a `const` member function shall not modify the object and its non-static data members~~. **—*end note*]** [*Example:*...

---

675. Signedness of bit-field with typedef or template parameter type

Section: 12.2.4 [class.bit] **Status:** CD3 **Submitter:** Richard Corden **Date:** 11 February, 2008

[Moved to DR at the October, 2012 meeting.]

Is the signedness of `x` in the following example implementation-defined?

```
template <typename T> struct A {
    T x : 7;
};

template struct A<long>;
```

A similar example could be created with a typedef.

Lawrence Crowl: According to 12.2.4 [class.bit] paragraph 3,

> It is implementation-defined whether a plain (neither explicitly signed nor unsigned) `char`, `short`, `int` or `long` bit-field is signed or unsigned.

This clause is conspicuously silent on typedefs and template parameters.

Clark Nelson: At least in C, the intention is that the presence or absence of this redundant keyword is supposed to be remembered through typedef declarations. I don't remember discussing it in C++, but I would certainly hope that we don't want to do something different. And presumably, we would want template type parameters to work the same way.

So going back to the original example, in an instantiation of `A<long>`, the signedness of the bit-field is implementation-defined, but in an instantiation of `A<signed long>`, the bit-field is definitely signed.

Peter Dimov: How can this work? Aren't `A<long>` and `A<signed long>` the same type?

(See also issue 739.)

Proposed resolution (February, 2012):

This issue is resolved by the resolution of issue 739.

---

739. Signedness of plain bit-fields

Section: 12.2.4 [class.bit] **Status:** CD3 **Submitter:** Mike Miller **Date:** 3 November, 2008

[Moved to DR at the October, 2012 meeting.]

12.2.4 [class.bit] paragraph 3 says,

> It is implementation-defined whether a plain (neither explicitly signed nor unsigned) `char`, `short`, `int` or `long` bit-field is signed or unsigned.

The implications of this permission for an implementation that chooses to treat plain bit-fields as unsigned are not clear. Does this mean that the type of such a bit-field is adjusted to the unsigned variant or simply that sign-extension is not performed when the value is fetched? C99 is explicit in specifying the former (6.7.2 paragraph 5: "for bit-fields, it is implementation-defined whether the specifier `int` designates the same type as `signed int` or the same type as `unsigned int`"), while C90 takes the latter approach (6.5.2.1: "Whether the high-order bit position of a (possibly qualified) 'plain' int bit-field is treated as a sign bit is implementation-defined").

(See also issue 675 and issue 741.)

Additional note, May, 2009:

As an example of the implications of this question, consider the following declaration:

```
struct S {
  int i: 2;
  signed int si: 2;
  unsigned int ui: 2;
} s;
```

Is it implementation-defined which expression, `cond?s.i:s.si` or `cond?s.i:s.ui`, is an lvalue (the lvalueness of the result depends on the second and third operands having the same type, per 8.16 [expr.cond] paragraph 4)?

Proposed resolution (August, 2011):

Change 12.2.4 [class.bit] paragraph 3 as follows:

> A bit-field shall not be a static member. A bit-field shall have integral or enumeration type (6.9.1 [basic.fundamental]). ~~It is implementation-defined whether a plain (neither explicitly signed nor unsigned) `char`, `short`, `int`, `long`, or `long long` bit-field is signed or unsigned.~~ **For a bit-field with a non-dependent type (17.7.2.1 [temp.dep.type]) that is specified to be plain (neither explicitly signed nor unsigned) `short`, `int`, `long`, or `long long` or a *typename-name* that is so defined (possibly through multiple levels of `typedef`s), it is implementation-defined whether the type of the bit-field is the corresponding signed or unsigned type. [*Example:***
>
> ```
> struct B {
>   long x : 3;
>   typedef signed int si;
>   si y : 1;
>   typedef int i;
>   i z : 1;
> };
>
> template<class T>
> struct A {
>   T x : 7;
> };
> ```
>
> **It is implementation-defined whether `B::x` has type `signed long` or `unsigned long`. `B::y` has type `signed int`. It is implementation-defined whether `B::z` has type `signed int` or `unsigned int`. `A<int>::x` and `A<signed int>::x` designate the same entity of type `signed int`. `A<unsigned int>::x` has type `unsigned int`. —*end example*]**
>
> A `bool` value...

This resolution also resolves issue 675.

Note, January, 2012:

Additional questions have been raised about the proposed resolution, so the status was returned to "review" to allow further discussion.

Proposed resolution (February, 2012):

1. Change 12.2.4 [class.bit] paragraph 3 as follows:

A bit-field shall not be a static member. A bit-field shall have integral or enumeration type (6.9.1 [basic.fundamental]). ~~It is implementation-defined whether a plain (neither explicitly signed nor unsigned)~~ `char`, ~~`short`~~, `int`, ~~`long`~~, or ~~`long long`~~ ~~bit-field is signed or unsigned.~~ A `bool` value can successfully be stored...

2. Add the following as a new section in C.1.8 [diff.class]:

12.2.4 [class.bit]

Change: Bit-fields of type plain `int` are signed.

Rationale: Leaving the choice of signedness to implementations could lead to inconsistent definitions of template specializations. For consistency, the implementation freedom was eliminated for non-dependent types, too.

Effect on original feature: The choice is implementation-defined in C, but not so in C++.

Difficulty of converting: Syntactic transformation.

How widely used: Seldom.

This resolution also resolves issue 675.

1375. Reference to anonymous union?

Section: 12.3 [class.union] **Status:** CD3 **Submitter:** Daveed Vandevoorde **Date:** 2011-08-16

[Moved to DR at the October, 2012 meeting.]

According to 12.3 [class.union] paragraph 7,

> A union for which objects or pointers are declared is not an anonymous union.

This should also apply to references, which are now possible because `decltype` allows writing an initializer for an unnamed union:

```
char buf[100];
union {
  int i;
} &r = (decltype(r)) buf;
```

Proposed resolution (February, 2012):

Change 12.3 [class.union] paragraph 7:

> A union for which objects**,** ~~or~~ pointers**, or references** are declared is not an anonymous union. [*Example:*
>
> ```
> void f() {
>   union { int aa; char* p; } obj, *ptr = &obj;
>   aa = 1;      // error
>   ptr->aa = 1; // OK
> }
> ```

1250. Cv-qualification of incomplete virtual function return types

Section: 13.3 [class.virtual] **Status:** CD3 **Submitter:** Jonathan Wakely **Date:** 2011-03-03

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

According to 13.3 [class.virtual] paragraph 8,

> If the return type of `D::f` differs from the return type of `B::f`, the class type in the return type of `D::f` shall be complete at the point of declaration of `D::f` or shall be the class type `D`.

This provision was intended to deal with covariant return types but inadvertently affects types that vary only in cv-qualification:

```
struct A;
struct B {
    virtual const A* f();
};
struct D : B {
    A* f();    // ill-formed
};
```

Proposed resolution (August, 2011):

Change 13.3 [class.virtual] paragraph 8 as follows:

> If the **class type in the covariant** return type of `D::f` differs from ~~the return type~~ **that** of `B::f`, the class type in the return type of `D::f` shall be complete at the point of declaration of `D::f` or shall be the class type `D`. When the overriding function...

---

462. Lifetime of temporaries bound to comma expressions

Section: 15.2 [class.temporary] **Status:** CD3 **Submitter:** Steve Adamczyk **Date:** April 2004

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

Split off from issue 86.

Should binding a reference to the result of a "," operation whose second operand is a temporary extend the lifetime of the temporary?

```
const SFileName &C = ( f(), SFileName("abc") );
```

Notes from the March 2004 meeting:

We think the temporary should be extended.

Proposed resolution (October, 2004):

Change 15.2 [class.temporary] paragraph 2 as indicated:

> ... In all these cases, the temporaries created during the evaluation of the expression initializing the reference, except the temporary **that is the overall result of the expression [*Footnote:* For example, if the expression is a comma expression (8.19 [expr.comma]) and the value of its second operand is a temporary, the reference is bound to that temporary.] and** to which the reference is bound, are destroyed at the end of the full-expression in which they are created and in the reverse order of the completion of their construction...

[Note: this wording partially resolves issue 86. See also issue 446.]

Notes from the April, 2005 meeting:

The CWG suggested a different approach from the 10/2004 resolution, leaving 15.2 [class.temporary] unchanged and adding normative wording to 8.19 [expr.comma] specifying that, if the result of the second operand is a temporary, that temporary is the result of the comma expression as well.

Proposed Resolution (November, 2006):

Add the indicated wording to 8.19 [expr.comma] paragraph 1:

> ... The type and value of the result are the type and value of the right operand; the result is of the same value category as its right operand, and is a bit-field if its right operand is a glvalue and a bit-field. **If the value of the right operand is a temporary (15.2 [class.temporary]), the result is that temporary.**

1336. Definition of "converting constructor"

Section: 15.3.1 [class.conv.ctor] **Status:** CD3 **Submitter:** Niels Dekker **Date:** 2011-07-03

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

Although the normative wording of 15.3.1 [class.conv.ctor] paragraph 1 defining a converting constructor says

> A constructor declared without the *function-specifier* `explicit` specifies a conversion from the types of its parameters to the type of its class.

implying that a constructor with multiple parameters can be a converting constructor, it would be helpful if the example contained such a constructor.

Proposed resolution (August, 2011):

1. Change the example in 15.3.1 [class.conv.ctor] paragraph 1 as follows:

```
struct X {
    X(int);
    X(const char*, int =0);
     X(int, int);
};

void f(X arg) {
  X a = 1;          // a = X(1)
  X b = "Jessie";   // b = X("Jessie",0)
  a = 2;            // a = X(2)
  f(3);             // f(X(3))
   f({1, 2});         // f(X(1,2))
}
```

2. Change the example in 15.3.1 [class.conv.ctor] paragraph 2 as follows:

```
struct Z {
  explicit Z();
```

```
  explicit Z(int);
   explicit Z(int, int);
};

Z a;                        // OK: default-initialization performed
Z a1 = 1;                   // error: no implicit conversion
Z a3 = Z(1);                // OK: direct initialization syntax used
Z a2(1);                    // OK: direct initialization syntax used
Z* p = new Z(1);            // OK: direct initialization syntax used
Z a4 = (Z)1;                // OK: explicit cast used
Z a5 = static_cast<Z>(1);   // OK: explicit cast used
 Z a6 = { 3, 4 };               // error: no implicit conversion
```

---

344. Naming destructors

Section: 15.4 [class.dtor] **Status:** CD3 **Submitter:** Jamie Schmeiser **Date:** 25 April 2002

Note that destructors suffer from similar problems as those of constructors dealt with in issue 194 and in 263 (constructors as friends). Also, the wording in 15.4 [class.dtor], paragraph 1 does not permit a destructor to be defined outside of the memberlist.

Change 15.4 [class.dtor], paragraph 1 from

> ...A special declarator syntax using an optional *function-specifier* (10.1.2 [dcl.fct.spec]) followed by ~ followed by the destructor's class name followed by an empty parameter list is used to declare the destructor in a class definition. In such a declaration, the ~ followed by the destructor's class name can be enclosed in optional parentheses; such parentheses are ignored....

to

> ...A special declarator syntax using an optional sequence of *function-specifier*s (10.1.2 [dcl.fct.spec]), an optional friend keyword, an optional sequence of *function-specifier*s (10.1.2 [dcl.fct.spec]) followed by an optional :: scope-resolution-operator followed by an optional *nested-name-specifier* followed by ~ followed by the destructor's class name followed by an empty parameter list is used to declare the destructor. The optional *nested-name-specifier* shall not be specified in the declaration of a destructor within the member-list of the class of which the destructor is a member. In such a declaration, the optional :: scope-resolution-operator followed by an optional *nested-name-specifier* followed by ~ followed by the destructor's class name can be enclosed in optional parentheses; such parentheses are ignored....

Proposed resolution:

This issue is resolved by the resolution of issue 1435.

1605. Misleading parenthetical comment for explicit destructor call

Section: 15.4 [class.dtor] **Status:** CD3 **Submitter:** Mike Miller **Date:** 2013-01-13

[Moved to DR at the April, 2013 meeting.]

According to 15.4 [class.dtor] paragraph 13,

> The invocation of a destructor is subject to the usual rules for member functions (12.2.1 [class.mfct]), that is, if the object is not of the destructor's class type and not of a class derived from the destructor's class type, the program has undefined behavior (except that invoking `delete` on a null pointer has no effect).

While true, the final parenthetical comment concerns a *delete-expression*, not an explicit destructor call. Its presence here could mislead a careless reader to think that invoking a destructor with a null pointer is harmless. It should be deleted.

Proposed resolution (February, 2013):

Change 15.4 [class.dtor] paragraph 13 as follows:

> In an explicit destructor call, the destructor name appears as a ~ followed by a *type-name* or *decltype-specifier* that denotes the destructor's class type. The invocation of a destructor is subject to the usual rules for member functions (12.2.1 [class.mfct])**:;** that is, if the object is not of the destructor's class type and not of a class derived from the destructor's class type **(including when the destructor is invoked via a null pointer value)**, the program has undefined behavior ~~(except that invoking delete on a null pointer has no effect)~~. **[*Note:* invoking `delete` on a null pointer does not call the destructor; see 8.3.5 [expr.delete]. — *end note*]** [*Example:*...

1345. Initialization of anonymous union class members

Section: 15.6.2 [class.base.init] **Status:** CD3 **Submitter:** Sean Hunt **Date:** 2011-08-16

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

15.6.2 [class.base.init] paragraph 8 appears to indicate that a class member that is an anonymous union is to be default initilized.

Proposed resolution (August, 2011):

Change the indicated bullet of 15.6.2 [class.base.init] paragraph 8 as follows:

- otherwise, if the entity is **an anonymous union or** a variant member (12.3 [class.union]), no initialization is performed;

---

535. Copy construction without a copy constructor

Section: 15.8 [class.copy] **Status:** CD3 **Submitter:** Mike Miller **Date:** 7 October 2005

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

Footnote 112 (15.8 [class.copy] paragraph 2) says,

> Because a template constructor is never a copy constructor, the presence of such a template does not suppress the implicit declaration of a copy constructor. Template constructors participate in overload resolution with other constructors, including copy constructors, and a template constructor may be used to copy an object if it provides a better match than other constructors.

However, many of the stipulations about copy construction are phrased to refer only to "copy constructors." For example, 15.8 [class.copy] paragraph 14 says,

> A program is ill-formed if the copy constructor... for an object is implicitly used and the special member function is not accessible (clause 14 [class.access]).

Does that mean that using an inaccessible template constructor to copy an object is permissible, because it is not a "copy constructor?" Obviously not, but each use of the term "copy constructor" in the Standard should be examined to determine if it applies strictly to copy constructors or to any constructor used for copying. (A similar issue applies to "copy assignment operators," which have the same relationship to assignment operator function templates.)

Proposed Resolution (August, 2011):

1. Change 6.2 [basic.def.odr] paragraph 2 as follows:

   > ...[*Note:* This covers calls to named functions (8.2.2 [expr.call]), operator overloading (Clause 16 [over]), user-defined conversions (15.3.2 [class.conv.fct]), allocation function for placement new (8.3.4 [expr.new]), as well as non-default initialization (11.6 [dcl.init]). A ~~copy constructor or move~~ constructor **selected to copy or move an object of class type** is odr-used even if the call is actually elided by the implementation **(15.8 [class.copy])**. —*end note*] ... ~~A copy assignment function for a class~~ **An assignment operator function in a class** is odr-used by an implicitly-defined copy-assignment **or move-assignment** function for another class as specified in 15.8 [class.copy]. ~~A move assignment function for a class is odr-used by an implicitly-defined move-assignment function for another class as specified in 15.8 [class.copy].~~ A default constructor...

2. Delete 15.1 [class.ctor] paragraph 9:

   > ~~A copy constructor (15.8 [class.copy]) is used to copy objects of class type. A move constructor (15.8 [class.copy]) is used to move the contents of objects of class type.~~

3. Change 15.2 [class.temporary] paragraph 1 as follows:

   > ...[*Note:* ~~even if there is no call to the destructor or copy/move constructor, all the semantic restrictions, such as accessibility (Clause 14 [class.access]) and whether the function is deleted (11.4.3 [dcl.fct.def.delete]), shall be satisfied.~~ **this includes accessibility (14 [class.access]) and whether it is deleted, for the constructor selected and for the destructor.** However, in the special case of a function call...

4. Change 15.8 [class.copy] paragraph 13 as follows:

   > A copy/move constructor that is defaulted and not defined as deleted is implicitly defined if it is odr-used (6.2 [basic.def.odr]) ~~to initialize an object of its class type from a copy of an object of its class type or of a class type derived from its class type [*Footnote:* See 11.6 [dcl.init] for more details on direct and copy initialization. —*end footnote*]~~ or when it is explicitly defaulted...

5. Change 15.8 [class.copy] paragraph 31 as follows:

   > When certain criteria are met, an implementation is allowed to omit the copy/move construction of a class object, even if the ~~copy/move~~ constructor **selected for the copy/move operation** and/or **the** destructor for the object have side effects...

6. Change 16.3.3.1.2 [over.ics.user] paragraph 4 as follows:

   > A conversion of an expression of class type to the same class type is given Exact Match rank, and a conversion of an expression of class type to a base class of that type is given Conversion rank, in spite of the fact that a ~~copy/move~~

constructor (i.e., a user-defined conversion function) is called for those cases.

7. Change 18.1 [except.throw] paragraph 3 as follows:

   A *throw-expression* **copy-**initializes **(11.6 [dcl.init])** a temporary object, called the *exception object*...

8. Change 18.1 [except.throw] paragraph 5 as follows:

   When the thrown object is a class object, the ~~copy/move~~ constructor **selected for the copy-initialization** and the destructor shall be accessible, even if the copy/move operation is elided (15.8 [class.copy]).

[Drafting note: 8.18 [expr.ass] paragraph 4, 12 [class] paragraph 4, 12.3 [class.union] paragraph 1, 15.2 [class.temporary] paragraph 2, 15.8 [class.copy] paragraphs 1-2, and 18.4 [except.spec] paragraph 14 do not require any changes.]

1402. Move functions too often deleted

Section: 15.8 [class.copy] **Status:** CD3 **Submitter:** Daniel Krügler **Date:** 2011-10-03

[Moved to DR status at the April, 2013 meeting as paper N3667.]

Paragraphs 11 and 23 of 15.8 [class.copy] make a defaulted move constructor and assignment operator, respectively, deleted if there is a subobject with no corresponding move function and for which the copy operation is non-trivial. This seems excessive and unnecessary. For example:

```
template<typename T>
 struct wrap
 {
  wrap() = default;

#ifdef USE_DEFAULTED_MOVE
  wrap(wrap&&) = default;
#else
  wrap(wrap&& w) : t(static_cast<T&&>(w.t)) { }
#endif

  wrap(const wrap&) = default;

  T t;
 };

struct S {
  S(){}
  S(const S&){}
  S(S&&){}
};

typedef wrap<const S> W;

W get() { return W(); }  // Error, if USE_DEFAULTED_MOVE is defined, else OK
```

In this example the defaulted move constructor of `wrap` is selected by overload resolution, but this move-constructor is deleted, because `S` has no *trivial* copy-constructor.

I think that we overshoot here with the delete rules: I see no problem for the defaulted move-constructor in this example. Our triviality-deduction rules already cover this case (15.8 [class.copy] paragraph 12: `W::W(W&&)` is not trivial) and our exception-specification rules (18.4 [except.spec] paragraph 14) already correctly deduce a `noexcept(false)` specification for `W::W(W&&)`.

It would still be OK to prevent that a move-constructor would be generated for the following example where no user-declared defaulted copy/move members are present:

```
template<typename T>
 struct wrap_2
 {
  wrap_2() = default;
  T t;
 };

typedef wrap_2<const S> W2;

W2 get() { return W2(); }  // OK, selects copy constructor
```

if we want. This would mean that we add a new bullet to 15.8 [class.copy] paragraph 9 and paragraph 20.

Proposed resolution (February, 2012):

1. Change 15.8 [class.copy] paragraph 9 as follows:

   If the definition of a class `X` does not explicitly declare a move constructor, one will be implicitly declared as defaulted if and only if

   - ...
   - X does not have a user-declared destructor, ~~and~~
   - the move constructor would not be implicitly defined as deleted~~.~~**, and**

- **each of `X`'s non-static data members and direct or virtual base classes has a type that either has a move constructor or is trivially copyable.**

[*Note:*...

2. Change 15.8 [class.copy] paragraph 11 as follows:

An implicitly-declared copy/move constructor is an `inline public` member of its class. A defaulted copy/move constructor for a class `X` is defined as deleted (11.4.3 [dcl.fct.def.delete]) if `X` has:

- ...
- any direct or virtual base class or non-static data member of a type with a destructor that is deleted or inaccessible from the defaulted constructor, **or**
- for the copy constructor, a non-static data member of rvalue reference type.~~, or~~
- ~~for the move constructor, a non-static data member or direct or virtual base class with a type that does not have a move constructor and is not trivially copyable.~~

3. Change 15.8 [class.copy] paragraph 20 as follows:

If the definition of a class `X` does not explicitly declare a move assignment operator, one will be implicitly declared as defaulted if and only if

- ...
- `X` does not have a user-declared destructor, ~~and~~
- the move assignment operator would not be implicitly defined as deleted~~.~~**,**
- `X` **has no direct or indirect virtual base class with a non-trivial move assignment operator, and**
- **each of `X`'s non-static data members and direct or virtual base classes has a type that either has a move assignment operator or is trivially copyable.**

Example:...

4. Change 15.8 [class.copy] paragraph 23 as follows:

A defaulted copy/move assignment operator for class `X` is defined as deleted if `X` has:

- ...
- a direct or virtual base class `B` that cannot be copied/moved because overload resolution (16.3 [over.match]), as applied to `B`'s corresponding assignment operator, results in an ambiguity or a function that is deleted or inaccessible from the defaulted assignment operator.~~, or~~
- ~~for the move assignment operator, a non-static data member or direct base class with a type that does not have a move assignment operator and is not trivially copyable, or any direct or indirect virtual base class.~~

Additional notes (August, 2012):

The proposed resolution was extensively discussed and additional alternatives were suggested. A paper is being produced for the October, 2012 meeting describing the various options, so the issue has been returned to "review" status to wait for the outcome of that deliberation.

See also the discussion of issue 1491 for additional considerations.

Proposed resolution (December, 2012):

1. Change 15.8 [class.copy] paragraph 9 as follows:

If the definition of a class `X` does not explicitly declare a move constructor, one will be implicitly declared as defaulted if and only if

- `X` does not have a user-declared copy constructor,
- `X` does not have a user-declared copy assignment operator,
- `X` does not have a user-declared move assignment operator, **and**
- `X` does not have a user-declared destructor.~~, and~~
- ~~the move constructor would not be implicitly defined as deleted.~~

[*Note:*...

2. Change 15.8 [class.copy] paragraph 11 as follows:

...A defaulted copy/move constructor for a class `X` is defined as deleted (11.4.3 [dcl.fct.def.delete]) if `X` has:

- a variant member with a non-trivial corresponding constructor and `X` is a union-like class,

- a non-static data member of class type `M` (or array thereof) that cannot be copied/moved because overload resolution (16.3 [over.match]), as applied to `M`'s corresponding constructor, results in an ambiguity or a function that is deleted or inaccessible from the defaulted constructor,
- a direct or virtual base class `B` that cannot be copied/moved because overload resolution (16.3 [over.match]), as applied to `B`'s corresponding constructor, results in an ambiguity or a function that is deleted or inaccessible from the defaulted constructor,
- any direct or virtual base class or non-static data member of a type with a destructor that is deleted or inaccessible from the defaulted constructor, **or**
- for the copy constructor, a non-static data member of rvalue reference type**.**~~, or~~
- ~~for the move constructor, a non-static data member or direct or virtual base class with a type that does not have a move constructor and is not trivially copyable.~~

A defaulted move constructor that is defined as deleted is ignored by overload resolution (16.3 [over.match]). [*Note:* A deleted move constructor would otherwise interfere with initialization from an rvalue which can use the copy constructor instead. —*end note*]

3. Change 15.8 [class.copy] paragraph 20 as follows:

If the definition of a class `X` does not explicitly declare a move assignment operator, one will be implicitly declared as defaulted if and only if

- `X` does not have a user-declared copy constructor,
- `X` does not have a user-declared move constructor,
- `X` does not have a user-declared copy assignment operator, **and**
- `X` does not have a user-declared destructor**.**~~, and~~
- ~~the move assignment operator would not be implicitly defined as deleted.~~

[*Example:*...

4. Change 15.8 [class.copy] paragraph 23 as follows:

A defaulted copy/move assignment operator for class `X` is defined as deleted if `X` has:

- a variant member with a non-trivial corresponding assignment operator and `X` is a union-like class, or
- a non-static data member of const non-class type (or array thereof), or
- a non-static data member of reference type, or
- a non-static data member of class type `M` (or array thereof) that cannot be copied/moved because overload resolution (16.3 [over.match]), as applied to `M`'s corresponding assignment operator, results in an ambiguity or a function that is deleted or inaccessible from the defaulted assignment operator, or
- a direct or virtual base class `B` that cannot be copied/moved because overload resolution (16.3 [over.match]), as applied to `B`'s corresponding assignment operator, results in an ambiguity or a function that is deleted or inaccessible from the defaulted assignment operator**.**~~, or~~
- ~~for the move assignment operator, a non-static data member or direct base class with a type that does not have a move assignment operator and is not trivially copyable, or any direct or indirect virtual base class.~~

A defaulted move assignment operator that is defined as deleted is ignored by overload resolution (16.3 [over.match], 16.4 [over.over]).

This resolution also resolves issue 1491.

1491. Move construction and rvalue reference members

Section: 15.8 [class.copy] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2012-03-30

According to 15.8 [class.copy] paragraph 11, the last bullet, a defaulted move constructor for a class is defined as deleted if

> a non-static data member or direct or virtual base class with a type that does not have a move constructor and is not trivially copyable.

This means that an example like

```
struct S { S(); int &&r; } s{S()};
```

is ill-formed. This is probably not intended.

(Note that the February, 2012 proposed resolution for issue 1402 also makes this example ill-formed, because the move constructor is not declared and the copy constructor is defined as deleted because of the rvalue-reference member.)

Additional note, February, 2014:

This issue is resolved by the resolution of issue 1402, which removed the problematic sentence.

1385. Syntactic forms of conversion functions for surrogate call functions

Section: 16.3.1.2 [over.match.oper] **Status:** CD3 **Submitter:** Daniel Krügler **Date:** 2011-08-31

[Moved to DR at the October, 2012 meeting.]

In 16.3.1.1.2 [over.call.object] paragraph 2, the non-explicit conversion functions considered for producing surrogate call functions are those of the form

`operator` *conversion-type-id* `()` *attribute-specifier-seq*$_{opt}$ *cv-qualifier* `;`

This (presumably inadvertently) excludes conversion functions with a *ref-qualifier* and/or an *exception-specification*.

Proposed resolution (February, 2012):

Change 16.3.1.1.2 [over.call.object] paragraph 2 as follows:

> In addition, for each non-explicit conversion function declared in `T` of the form
>
> `operator` *conversion-type-id* `()` ~~*attribute-specifier-seq*$_{opt}$~~ *cv-qualifier*
> ***ref-qualifier*$_{opt}$ *exception-specification*$_{opt}$ *attribute-specifier-seq*$_{opt}$** `;`
>
> where *cv-qualifier* is the same...

1556. Constructors and explicit conversion functions in direct initialization

Section: 16.3.1.4 [over.match.copy] **Status:** CD3 **Submitter:** Edward Catmur **Date:** 2012-09-18

[Moved to DR at the April, 2013 meeting.]

Issue 899 added wording to the second bullet of 16.3.1.4 [over.match.copy] paragraph 1 permitting use of `explicit` conversion functions when calling a copy constructor in a direct-initialization context. Issue 1087 addressed the problem in the earlier resolution that the phrase "copy constructor" did not include move constructors and template constructors. However, the term "copy constructor" implicitly (and correctly) restricted the constructor parameter to be the constructor's class, i.e., the class of the object being directly initialized. The new phrasing, "a constructor that takes a reference to possibly cv-qualified `T` as its first argument," incorrectly assumed that `T` referred to the type of the object being initialized; however, that is not the case, and a converting constructor from `T` to the type of the object being initialized is inadvertently permitted. The wording needs a further tweak to restore the intended context.

Proposed resolution (October, 2012):

Change the second bullet of 16.3.1.4 [over.match.copy] paragraph 1 as follows:

> ...Assuming that "*cv1* `T`" is the type of the object being initialized, with `T` a class type, the candidate functions are selected as follows:
>
> - The converting constructors (15.3.1 [class.conv.ctor]) of `T` are candidate functions.
> - When the type of the initializer expression is a class type "*cv* `S`", the non-explicit conversion functions of `S` and its base classes are considered. When initializing a temporary to be bound to the first parameter of a constructor that takes a reference to possibly cv-qualified `T` as its first argument, called with a single argument in the context of direct-initialization **of an object of type "*cv2* `T`"**, explicit conversion functions are also considered. Those that are not hidden within `S` and yield a type whose cv-unqualified version is the same type as `T` or is a derived class thereof are candidate functions. Conversion functions that return "reference to `X`" return lvalues or xvalues, depending on the type of reference, of type `X` and are therefore considered to yield `X` for this process of selecting candidate functions.

1392. Explicit conversion functions for references and non-references

Section: 16.3.1.6 [over.match.ref] **Status:** CD3 **Submitter:** Jason Merrill **Date:** 2011-09-08

[Moved to DR at the October, 2012 meeting.]

In 16.3.1.5 [over.match.conv], dealing with non-reference initialization, direct initialization considers as candidate functions only those that

yield type T or a type that can be converted to type T with a qualification conversion

By contrast, 16.3.1.6 [over.match.ref], dealing with reference binding, requires only that the type returned be reference-compatible with the target, permitting both qualification conversions and derived-to-base conversions. This discrepancy is presumably unintentional.

Proposed resolution (February, 2012):

Change 16.3.1.6 [over.match.ref] paragraph 1 as follows:

...the candidate functions are selected as follows:

- The conversion functions of S and its base classes are considered~~, except that for copy-initialization, only the non-explicit conversion functions are considered~~. Those **non-explicit conversion functions** that are not hidden within S and yield type "lvalue reference to *cv2* T2" (when 11.6.3 [dcl.init.ref] requires an lvalue result) or "*cv2* T2" or "rvalue reference to *cv2* T2" (when 11.6.3 [dcl.init.ref] requires an rvalue result), where "*cv1* T" is reference-compatible (11.6.3 [dcl.init.ref]) with "*cv2* T2", are candidate functions. **For direct-initialization, those explicit conversion functions that are not hidden within S and yield type "lvalue reference to *cv2* T2," or "*cv2* T2*#8221; or "rvalue reference to *cv2* T2," respectively, where T2 is the same type as T or can be converted to type T with a qualification conversion (7.5 [conv.qual]), are also candidate functions.**

1409. What is the second standard conversion sequence of a list-initialization sequence?

Section: 16.3.3.1.5 [over.ics.list] **Status:** CD3 **Submitter:** Sebastian Redl **Date:** 2011-10-24

[Moved to DR at the October, 2012 meeting.]

Both paragraphs 3 and 4 (for non-aggregate and aggregate types, respectively) of 16.3.3.1.5 [over.ics.list] say that the implicit conversion sequence is a user-defined conversion sequence, but neither specifies that the second standard conversion sequence is the identity conversion, as is presumably intended. This makes ranking by 16.3.3.2 [over.ics.rank] paragraph 3 bullet 2 unncessarily unclear.

Proposed resolution (February, 2012):

Change 16.3.3.1.5 [over.ics.list] paragraphs 3-4 as follows:

> Otherwise, if the parameter is a non-aggregate class X and overload resolution per 16.3.1.7 [over.match.list] chooses a single best constructor of X to perform the initialization of an object of type X from the argument initializer list, the implicit conversion sequence is a user-defined conversion sequence **with the second standard conversion sequence an identity conversion**. If multiple constructors are viable but none is better than the others, the implicit conversion sequence is the ambiguous conversion sequence...
>
> Otherwise, if the parameter has an aggregate type which can be initialized from the initializer list according to the rules for aggregate initialization (11.6.1 [dcl.init.aggr]), the implicit conversion sequence is a user-defined conversion sequence **with the second standard conversion sequence an identity conversion**. [*Example*:...

1543. Implicit conversion sequence for empty initializer list

Section: 16.3.3.1.5 [over.ics.list] **Status:** CD3 **Submitter:** Steve Adamczyk **Date:** 2012-08-21

[Moved to DR at the April, 2013 meeting.]

According to 16.3.3.1.5 [over.ics.list] paragraph 6, when passing an initializer-list argument to a non-class parameter,

> if the initializer list has no elements, the implicit conversion sequence is the identity conversion.

However, there is no similar provision for an empty initializer list passed to a specialization of `std::initializer_list` or an array, as described in paragraph 2:

> If the parameter type is `std::initializer_list<X>` or "array of X" [134] and all the elements of the initializer list can be implicitly converted to X, the implicit conversion sequence is the worst conversion necessary to convert an element of the list to X.

It is not clear what the result should be for a list with no elements. For example, given

```
void f(int) { printf("int\n"); }
void f(std::initializer_list<int>) { printf("init list\n"); }
int main() {
    f({});
}
```

current implementations result in `init list` being printed, presumably on the basis of the last bullet of 16.3.3.2 [over.ics.rank] paragraph 3:

- List-initialization sequence L1 is a better conversion sequence than list-initialization sequence L2 if L1 converts to `std::initializer_list<X>` for some X\and L2 does not.

That would imply that both conversion sequences are the identity conversion, which is reasonable, but it should be stated clearly if that is the intent.

Proposed resolution (October, 2012):

Change 16.3.3.1.5 [over.ics.list] paragraph 2 as follows:

> If the parameter type is `std::initializer_list<X>` or "array of `X`" [134] and all the elements of the initializer list can be implicitly converted to `X`, the implicit conversion sequence is the worst conversion necessary to convert an element of the list to `X`**, or if the initializer list has no elements, the identity conversion**. This conversion can be a user-defined conversion even in the context of a call to an initializer-list constructor. [*Example:*
>
> ```
> void f(std::initializer_list<int>);
> f( {} );         // OK: f(initializer_list<int>) identity conversion
> f( {1,2,3} );    // OK: f(initializer_list<int>) identity conversion
> f( {'a','b'} ); // OK: f(initializer_list<int>) integral promotion
> f( {1.0} );      // error: narrowing
> ...
> ```

1298. Incorrect example in overload resolution

Section: 16.3.3.2 [over.ics.rank] **Status:** CD3 **Submitter:** Nikolay Ivchenkov **Date:** 2011-04-15

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

The following example appears in 16.3.3.2 [over.ics.rank] paragraph 3:

```
template<class T> int f(T&);
template<class T> int f(T&&);
void g();
int i1 = f(g); // calls f(T&)
```

This is not correct. Because of the special deduction rule for rvalue reference parameters in 17.9.2.1 [temp.deduct.call] paragraph 3 and the reference-collapsing rules of 11.3.2 [dcl.ref] paragraph 6, the parameter type for both will be `void(&)()`.

Proposed resolution (August, 2011):

Change the example in 16.3.3.2 [over.ics.rank] paragraph 3 bullet 1 sub-bullet 5 as follows:

```
~~template<class T> int f(T&);~~
~~template<class T> int f(T&&);~~
int f(void(&)());      // #1
int f(void(&&)());     // #2
void g();
int i1 = f(g);         // calls ~~f(T&)~~ #1
```

1374. Qualification conversion vs difference in reference binding

Section: 16.3.3.2 [over.ics.rank] **Status:** CD3 **Submitter:** Michael Wong **Date:** 2011-08-15

[Moved to DR at the April, 2013 meeting.]

The rule in 16.3.3.2 [over.ics.rank] paragraph 3 for ranking based on a difference in qualification conversion applies only if they "differ only in their qualification conversion".

It is unclear as to whether the property of being a reference binding is a factor in determining if there is a difference between conversion sequences. Notice that 16.3.3.1.4 [over.ics.ref] maps reference bindings to other forms of implicit conversion sequences, but does not state that the property of being a reference binding is preserved; however, 16.3.3.2 [over.ics.rank] has cases which depend on whether certain standard conversion sequences are reference bindings or not and on the specifics of the bindings.

In the following, picking `T2 &&` would bind an rvalue to an rvalue reference. Picking `T1 &` would bind an rvalue to an lvalue reference, but the qualification conversion to `T1` is "better". Which is better?

```
typedef int *      *      *const *const T1;
typedef int *const *const *const *const T2;
void foo(T1 &);
void foo(T2 &&) { }

int main() {
   foo((int ****)0);
   return 0;
}
```

Notes from the February, 2012 meeting:

The CWG agreed that bullets 3 and 4 should be reversed, to check the reference binding first and then for qualification conversion.

Proposed resolution (February, 2012):

Move 16.3.3.2 [over.ics.rank] paragraph 3, first bullet, third sub-bullet, after the current fifth sub-bullet, as follows:

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

- Standard conversion sequence S1 is a better conversion sequence...
  - S1 is a proper subsequence of S2...
  - the rank of S1 is better...
  - ~~S1 and S2 differ only in their qualification conversion... —*end example*] or if not that,~~
  - S1 and S2 are reference bindings (11.6.3 [dcl.init.ref]) and neither refers... or if not that,
  - S1 and S2 are reference bindings (11.6.3 [dcl.init.ref]) and S1 binds... —*end example*] **or if not that,**
  - S1 **and** S2 **differ only in their qualification conversion... —*end example*] or if not that,**
  - S1 and S2 are reference bindings (11.6.3 [dcl.init.ref]), and the types to which the references refer...
- User-defined conversion sequence U1...

1408. What is "the same aggregate initialization?"

Section: 16.3.3.2 [over.ics.rank] **Status:** CD3 **Submitter:** Sebastian Redl **Date:** 2011-10-24

[Moved to DR at the October, 2012 meeting.]

Bullet 2 of 16.3.3.2 [over.ics.rank] paragraph 3 reads,

- User-defined conversion sequence U1 is a better conversion sequence than another user-defined conversion sequence U2 if they contain the same user-defined conversion function or constructor or aggregate initialization and the second standard conversion sequence of U1 is better than the second standard conversion sequence of U2.

It is not clear what "the same aggregate initialization" means — does this require that the same aggregate type is the target type?

Proposed resolution (February, 2012):

Change 16.3.3.2 [over.ics.rank] paragraph 3 bullet 2 as follows:

- Standard conversion sequence S1 is a better conversion sequence...
- User-defined conversion sequence U1 is a better conversion sequence than another user-defined conversion sequence U2 if they contain the same user-defined conversion function or constructor or **they initialize the same class in an** aggregate initialization and **in either case** the second standard conversion sequence of U1 is better than the second standard conversion sequence of U2. [*Example:*...

1410. Reference overload tiebreakers should apply to rvalue references

Section: 16.3.3.2 [over.ics.rank] **Status:** CD3 **Submitter:** Michael Wong **Date:** 2011-10-26

[Moved to DR at the October, 2012 meeting.]

In bullet 3 of paragraph 4 of 16.3.3.2 [over.ics.rank] are two sub-bullets dealing with overload tiebreakers:

- binding of an expression of type C to a reference of type B& is better than binding an expression of type C to a reference of type A&,
- ...
- binding of an expression of type B to a reference of type A& is better than binding an expression of type C to a reference of type A&,

Presumably both of these tiebreakers should apply to rvalue references as well as lvalue references.

Proposed resolution (February, 2012):

Change 16.3.3.2 [over.ics.rank] paragraph 4 bullet 3 as follows:

- If class B is derived directly or indirectly from class A and class C is derived directly or indirectly from B,
  - conversion of C* to B* is better...
  - binding of an expression of type C to a reference ~~of~~ **to** type B~~&~~ is better than binding an expression of type C to a reference ~~of~~ **to** type A~~&~~,
  - ...

- binding of an expression of type `B` to a reference ~~**of**~~ **to** type `A&` is better than binding an expression of type `C` to a reference ~~**of**~~ **to** type `A&`,
- ...

1563. List-initialization and overloaded function disambiguation

Section: 16.4 [over.over] **Status:** CD3 **Submitter:** Daniel Krügler **Date:** 2012-10-08

[Moved to DR at the April, 2013 meeting.]

Presumably, 16.4 [over.over] paragraph 1,

> A use of an overloaded function name without arguments is resolved in certain contexts to a function, a pointer to function or a pointer to member function for a specific function from the overload set. A function template name is considered to name a set of overloaded functions in such contexts. The function selected is the one whose type is identical to the function type of the target type required in the context. [*Note:* That is, the class of which the function is a member is ignored when matching a pointer-to-member-function type. —*end note*] The target can be
>
> - an object or reference being initialized (11.6 [dcl.init], 11.6.3 [dcl.init.ref]),
> - ...

should apply to an example like

```
double bar(double) { return 0.0; }
float bar(float) { return 0.0f; }

using fun = double(double);
fun &foo{bar};
```

However, there is implementation variance in whether the use of `bar` is accepted or not, and the omission of a cross-reference to 11.6.4 [dcl.init.list] might be read as indicating that list-initialization is not included.

Proposed resolution (February, 2013):

Change 16.4 [over.over] paragraph 1 as follows:

> ...The target can be
>
> - an object or reference being initialized (11.6 [dcl.init], 11.6.3 [dcl.init.ref]**, 11.6.4 [dcl.init.list]**),
> - ...

1481. Increment/decrement operators with reference parameters

Section: 16.5.7 [over.inc] **Status:** CD3 **Submitter:** Ryou Ezoe **Date:** 2012-03-20

[Moved to DR at the April, 2013 meeting.]

The phrase in 16.5.7 [over.inc] paragraph 1,

> a non-member function with one parameter of class or enumeration type

inadvertently excludes reference parameters, and in fact, no mention of the type is necessary in light of 16.5 [over.oper] paragraph 6:

> An operator function shall either be a non-static member function or be a non-member function and have at least one parameter whose type is a class, a reference to a class, an enumeration, or a reference to an enumeration.

Proposed resolution (August, 2012):

Change 16.5.7 [over.inc] paragraph 1 as follows:

> The user-defined function called `operator++` implements the prefix and postfix `++` operator. If this function is a member function with no parameters, or a non-member function with one parameter ~~of class or enumeration type~~, it defines the prefix increment operator ++ for objects of that type. If the function...

1473. Syntax of *literal-operator-id*

Section: 16.5.8 [over.literal] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2012-03-05

[Moved to DR at the April, 2013 meeting.]

The current grammar requires that there be no whitespace between the literal and the *ud-suffix*, e.g., `""_abc`, but that there be whitespace between the `""` and the *identifier* in a *literal-operator-id*, e.g., `operator "" _abc`. This seems unfortunate. Would it be possible to provide an alternate production,

`operator` *user-defined-string-literal*

with the requirement that the *string-literal* be empty?

The current wording is also unclear regarding interactions with phases of translation. We have the following production:

literal-operator-id:
`operator ""` *identifier*

As this is after translation phase 6, would something like

```
operator "" "" _foo
```

be accepted?

Proposed resolution (October, 2012):

1. Change 16.5.8 [over.literal] as follows:

literal-operator-id:
`operator ""` ***string-literal*** *identifier*
`operator` ***user-defined-string-literal***

2. Change 16.5.8 [over.literal] paragraph 1 as follows:

The *string-literal* or *user-defined-string-literal* in a *literal-operator-id* shall have no *encoding-prefix* and shall contain no characters other than the implicit terminating `'\0'`. The ***ud-suffix* of the *user-defined-string-literal* or the** *identifier* in a *literal-operator-id* is called a *literal suffix identifier*. [*Note:* some literal suffix identifiers are reserved for future standardization; see 20.5.4.3.5 [usrlit.suffix]. —*end note*]

3. Change the example in 16.5.8 [over.literal] paragraph 8 as follows:

```
void operator "" _km(long double);                  // OK
string operator "" _i18n(const char*, std::size_t); // OK
template <char...> int operator "" \u03C0();        // OK: UCN for lowercase pi
float operator ""E(const char*);                    // ~~error: ""E (with no intervening space)~~
                                                    // ~~is a single token~~ OK
float operator " " B(const char*);                  // error: non-~~adjacent quotes~~empty string-literal
string operator "" 5X(const char*, std::size_t);    // error: invalid literal suffix identifier
double operator "" _miles(double);                  // error: invalid parameter-declaration-clause
template <char...> int operator "" _j(const char*); // error: invalid parameter-declaration-clause
```

1479. Literal operators and default arguments

Section: 16.5.8 [over.literal] **Status:** CD3 **Submitter:** Jason Merrill **Date:** 2012-03-13

[Moved to DR at the April, 2013 meeting.]

It appears that an example like

```
int operator"" _a (const char *, std::size_t = 0);
int operator"" _a (const char *);

int i = 123_a;
```

is ambiguous: although only the second declaration is a raw literal operator, the corresponding call

```
operator"" _a ("123")
```

could match either declaration. Should default arguments for literal operators be prohibited?

Proposed resolution (October, 2012):

Change 16.5.8 [over.literal] paragraph 3 as follows:

The declaration of a literal operator shall have a *parameter-declaration-clause* equivalent to one of the following:

...

If a parameter has a default argument (11.3.6 [dcl.fct.default]), the program is ill-formed.

1275. Incorrect comment in example of template parameter pack restriction

Section: 17.1 [temp.param] **Status:** CD3 **Submitter:** Johannes Schaub **Date:** 2011-03-25

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

The following example from 17.1 [temp.param] paragraph 11 is incorrect:

```
// U cannot be deduced or specified
template<class... T, class... U> void f() { }
template<class... T, class U> void g() { }
```

In fact, `U` can be deduced to an empty sequence by 17.9.1 [temp.arg.explicit] paragraph 3:

> A trailing template parameter pack (17.6.3 [temp.variadic]) not otherwise deduced will be deduced to an empty sequence of template arguments.

Proposed resolution (August, 2011):

Change 17.1 [temp.param] paragraph 11 as follows:

> ...A template parameter pack of a function template shall not be followed by another template parameter unless that template parameter can be deduced **from the *parameter-type-list* of the function template** or has a default argument (17.9.2 [temp.deduct]). [*Example:*
>
> ```
> template<class T1 = int, class T2> class B;   // error
>
> // U cannot be deduced from the parameter-type-list or specified
> template<class... T, class... U> void f() { } // error
> template<class... T, class U> void g() { }    // error
> ```
>
> —*end example*]

1398. Non-type template parameters of type `std::nullptr_t`

Section: 17.3.2 [temp.arg.nontype] **Status:** CD3 **Submitter:** Daniel Krügler **Date:** 2011-09-27

[Moved to DR at the October, 2012 meeting.]

Although 17.1 [temp.param] paragraph 4 explicitly allows non-type template parameters of type `std::nullptr_t`, they cannot actually be used: 17.3.2 [temp.arg.nontype] paragraph 1 does not allow a *template-argument* of type `std::nullptr_t`, and paragraph 5 does not describe any conversions from other types to `std::nullptr_t`.

Proposed resolution (February, 2012):

Add the following new bullet in 17.3.2 [temp.arg.nontype] paragraph 1:

- ...
- a pointer to member expressed as described in 8.3.1 [expr.unary.op].
- **an address constant expression of type `std::nullptr_t`.**

1533. Function pack expansion for member initialization

Section: 17.6.3 [temp.variadic] **Status:** CD3 **Submitter:** Jonathan Caves **Date:** 2012-08-06

[Moved to DR at the April, 2013 meeting.]

The list of pack expansion contexts in 17.6.3 [temp.variadic] paragraph 4 includes a *mem-initializer-list*, with no restriction on whether the *mem-initializer* corresponds to a base class or a member. However, it appears from 15.6.2 [class.base.init] paragraph 15 that such a pack expansion is intended for bases:

> A *mem-initializer* followed by an ellipsis is a pack expansion (17.6.3 [temp.variadic]) that initializes the base classes specified by a pack expansion in the *base-specifier-list* for the class.

This is not conclusive, however, and use of a pack expansion with a *mem-initializer* for a member could be used with packs containing zero or one element:

```
class S { };

template<typename... T> class X {
public:
  X(T ...args) : data_(args)... { }
private:
  S data_;
};
```

```
int main() {
  S s;
  X<> x1;
  X<S> x2(s);
}
```

The Standard should be clarified as to whether such a usage is permitted or not.

Proposed resolution (October, 2012):

Change 17.6.3 [temp.variadic] paragraph 4 as follows:

...Pack expansions can occur in the following contexts:

- ...
- In a *mem-initializer-list* (15.6.2 [class.base.init]) **for a *mem-initializer* whose *mem-initializer-id* denotes a base class**; the pattern is ~~a~~ **the** *mem-initializer*.
- ...

1495. Partial specialization of variadic class template

Section: 17.6.5 [temp.class.spec] **Status:** CD3 **Submitter:** Jason Merrill **Date:** 2012-04-16

[Moved to DR at the April, 2013 meeting.]

Consider an example like

```
template <int B, typename Type1, typename... Types>
struct A;

template<typename... Types>
struct A<0, Types...> { };

A<0,int,int> t;
```

In this case, the partial specialization seems well-formed by the rules in 17.6.5 [temp.class.spec], but it is not more specialized than the primary template. However, 17.6.5.1 [temp.class.spec.match] says that if exactly one matching specialization is found, it is used, which suggests that the testcase is well-formed. That seems undesirable; I think a partial specialization that is not more specialized than the primary template should be ill-formed.

If the example is rewritten so that both versions are partial specializations, i.e.,

```
template <int B, typename... Types>
struct A;

template <int B, typename Type1, typename... Types>
struct A<B, Type1, Types...> { }

template<typename... Types>
struct A<0, Types...> { };

A<0,int,int> t;
```

There is implementation variance, with gcc and clang reporting an ambiguity and EDG choosing the second specialization.

Proposed resolution (October, 2012):

Add the following as a new bullet in 17.6.5 [temp.class.spec] paragraph 8:

- ...
- The argument list of the specialization shall not be identical to the implicit argument list of the primary template.
- **The specialization shall be more specialized than the primary template (17.6.5.2 [temp.class.order]).**
- ...

2035. Multi-section example is confusing

Section: 17.6.5.1 [temp.class.spec.match] **Status:** CD3 **Submitter:** CWG **Date:** 2014-11-06

The example in 17.6.5.1 [temp.class.spec.match] paragraph 2 reads,

```
A<int, int, 1>   a1;   // uses #1
A<int, int*, 1>  a2;   // uses #2, T is int, I is 1
A<int, char*, 5> a3;   // uses #4, T is char
```

```
A<int, char*, 1> a4;    // uses #5, T1 is int, T2 is char, I is 1
A<int*, int*, 2> a5;    // ambiguous: matches #3 and #5
```

However, the referenced numbers are defined two pages earlier, in 17.6.5 [temp.class.spec] paragraph 3. This is confusing and should be changed.

Notes from the May, 2015 meeting:

This will be handled editorially; the status has been set to "review" to check that the editorial change has been made.

1321. Equivalency of dependent calls

Section: 17.6.6.1 [temp.over.link] **Status:** CD3 **Submitter:** Jason Merrill **Date:** 2011-05-18

[Moved to DR at the October, 2012 meeting.]

Consider the following example:

```
int g(int);

template <class T> decltype(g(T())) f();

int g();

template <class T> decltype(g(T())) f() { return g(T()); }

int i = f<int>();
```

Do the two `f`s declare the same function template? According to 17.6.6.1 [temp.over.link] paragraph 5,

> Two expressions involving template parameters are considered *equivalent* if two function definitions containing the expressions would satisfy the one definition rule (6.2 [basic.def.odr]), except that the tokens used to name the template parameters may differ as long as a token used to name a template parameter in one expression is replaced by another token that names the same template parameter in the other expression.

The relevant portion of 6.2 [basic.def.odr] paragraph 5 says,

> in each definition of `D`, corresponding names, looked up according to 6.4 [basic.lookup], shall refer to an entity defined within the definition of `D`, or shall refer to the same entity, after overload resolution (16.3 [over.match]) and after matching of partial template specialization (17.9.3 [temp.over]), except that a name can refer to a const object with internal or no linkage if the object has the same literal type in all definitions of `D`, and the object is initialized with a constant expression (8.20 [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`

This could be read either way, since overload resolution isn't done at this point. Either we consider the result of the unqualified name lookup and say that the expressions aren't equivalent or we need a new rule for equivalence and merging of dependent calls.

Proposed resolution (December, 2011):

1. Change 17.6.6.1 [temp.over.link] paragraph 5 as follows:

> Two expressions involving template parameters are considered *equivalent* if two function definitions containing the expressions would satisfy the one definition rule (6.2 [basic.def.odr]), except that the tokens used to name the template parameters may differ as long as a token used to name a template parameter in one expression is replaced by another token that names the same template parameter in the other expression. **For determining whether two dependent names (17.7.2 [temp.dep]) are equivalent, only the name itself is considered, not the result of name lookup in the context of the template. If multiple declarations of the same function template differ in the result of this name lookup, the result for the first declaration is used.** [*Example:*
>
> ```
> template <int I, int J> void f(A<I+J>); // #1
> template <int K, int L> void f(A<K+L>); // same as #1
>
> template <class T> decltype(g(T())) h();
> int g(int);
> template <class T> decltype(g(T())) h() // redeclaration of h() uses the earlier lookup
>   { return g(T()); }                    // ...although the lookup here does find g(int)
> int i = h<int>();                       // template argument substitution fails; g(int)
>                                         // was not in scope at the first declaration of h()
> ```
>
> —*end example*] Two expressions...

2. Change 17.7.2 [temp.dep] paragraph 1 as follows:

> ...In an expression of the form:
>
> *postfix-expression* (*expression-list*$_{opt}$)
>
> where the *postfix-expression* is an ~~*id-expression*~~ ***unqualified-id***, the ~~*id-expression*~~ ***unqualified-id*** denotes a dependent name if
>
> - any of the expressions in the *expression-list* is a pack expansion (17.6.3 [temp.variadic]),

- any of the expressions in the *expression-list* is a type-dependent expression (17.7.2.2 [temp.dep.expr]), or
- if the *unqualified-id* ~~of the *id-expression*~~ is a *template-id* in which any of the template arguments depends on a template parameter.

if an operand...

3. Change 17.7.4.2 [temp.dep.candidate] paragraph 1 as follows:

For a function call ~~that depends on a template parameter~~ **where the *postfix-expression* is a dependent name**, the candidate functions are found using the usual lookup rules (6.4.1 [basic.lookup.unqual], 6.4.2 [basic.lookup.argdep]~~, 6.4.3 [basic.lookup.qual]~~) except that:

- For the part of the lookup using unqualified name lookup (6.4.1 [basic.lookup.unqual]) ~~or qualified name lookup (6.4.3 [basic.lookup.qual])~~, only function declarations from the template definition context are found.
- For the part of the lookup using associated namespaces (6.4.2 [basic.lookup.argdep]), only function declarations found in either the template definition context or the template instantiation context are found.

If ~~the function name is an *unqualified-id* and~~ the call would be ill-formed or would find a better match had the lookup within the associated namespaces considered all the function declarations with external linkage introduced in those namespaces in all translation units, not just considering those declarations found in the template definition and template instantiation contexts, then the program has undefined behavior.

1406. *ref-qualifier*s and added parameters of non-static member function templates

Section: 17.6.6.2 [temp.func.order] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2011-10-21

[Moved to DR at the October, 2012 meeting.]

In describing the partial ordering of function templates, 17.6.6.2 [temp.func.order] paragraph 3 says,

> If only one of the function templates is a non-static member, that function template is considered to have a new first parameter inserted in its function parameter list. The new parameter is of type "reference to *cv* A," where *cv* are the cv-qualifiers of the function template (if any) and A is the class of which the function template is a member. [*Note:* This allows a non-static member to be ordered with respect to a nonmember function and for the results to be equivalent to the ordering of two equivalent nonmembers. —*end note*]

The Standard appears to be silent as to whether the reference is an lvalue or rvalue reference; presumably that should be determined by the *ref-qualifier* of the member function, if any.

Proposed resolution (February, 2012):

Change 17.6.6.2 [temp.func.order] paragraph 3 as follows:

> To produce the transformed template, for each type, non-type, or template template parameter (including template parameter packs (17.6.3 [temp.variadic]) thereof) synthesize a unique type, value, or class template respectively and substitute it for each occurrence of that parameter in the function type of the template. If only one of the function templates is a non-static member **of some class** A, that function template is considered to have a new first parameter inserted in its function parameter list. ~~The~~ **Given *cv* as the cv-qualifiers of the function template (if any), the** new parameter is of type "**rvalue** reference to *cv* A~~,~~" **if the optional *ref-qualifier* of the function template is &&, or of type "lvalue reference to *cv* A" otherwise** ~~where *cv* are the cv-qualifiers of the function template (if any) and A is the class of which the function template is a member~~. [*Note:*...

1296. Ill-formed template declarations (not just definitions)

Section: 17.7 [temp.res] **Status:** CD3 **Submitter:** Nikolay Ivchenkov **Date:** 2011-04-14

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

According to 17.7 [temp.res] paragraph 8,

> Knowing which names are type names allows the syntax of every template definition to be checked. No diagnostic shall be issued for a template definition for which a valid specialization can be generated. If no valid specialization can be generated for a template definition, and that template is not instantiated, the template definition is ill-formed, no diagnostic required. If every valid specialization of a variadic template requires an empty template parameter pack, the template definition is ill-formed, no diagnostic required. If a type used in a non-dependent name...

It seems that these points could and should apply to template declarations that are not definitions, as well.

Proposed resolution (August, 2011):

Change 17.7 [temp.res] paragraph 8 as follows:

Knowing which names are type names allows the syntax of every template ~~definition~~ to be checked. No diagnostic shall be issued for a template ~~definition~~ for which a valid specialization can be generated. If no valid specialization can be generated for a template ~~definition~~, and that template is not instantiated, the template ~~definition~~ is ill-formed, no diagnostic required. If every valid specialization of a variadic template requires an empty template parameter pack, the template ~~definition~~ is ill-formed, no diagnostic required. If a type used...

1471. Nested type of non-dependent base

Section: 17.7.2.1 [temp.dep.type] **Status:** CD3 **Submitter:** Johannes Schaub **Date:** 2012-02-26

[Moved to DR at the April, 2013 meeting.]

Even though `A::C` is a nested type and member of the current instantiation, and thus dependent by the rules of 17.7.2.1 [temp.dep.type] paragraph 8, there does not seem to be a good reason for making it so:

```
struct B {
 struct C { };
};

template<typename T> struct A : B {
 A::C c;
};
```

Proposed resolution (October, 2012):

[Some existing uses of the term "member of the current instantiation" are consistent with the definition in 17.7.2.1 [temp.dep.type] paragraph 4; others are intended to refer to members of the "current instantiation," as defined in paragraph 1. The following resolution changes the latter to use the phrase "member of a class that is the current instantiation."]

1. Change 17.7.2.1 [temp.dep.type] paragraph 4 as follows:

   A name is a *member of the current instantiation* if it is

   - An unqualified name that, when looked up, refers to at least one member of **a class that is** the current instantiation or a non-dependent base class thereof. [*Note:* This can only occur when looking up a name in a scope enclosed by the definition of a class template. —*end note*]
   - A *qualified-id* in which the *nested-name-specifier* refers to the current instantiation and that, when looked up, refers to at least one member of **a class that is** the current instantiation or a non-dependent base class thereof. [*Note:* if no such member is found, and the current instantiation has any dependent base classes, then the *qualified-id* is a member of an unknown specialization; see below. —*end note*]
   - An *id-expression* denoting the member in a class member access expression (8.2.5 [expr.ref]) for which the type of the object expression is the current instantiation, and the *id-expression*, when looked up (6.4.5 [basic.lookup.classref]), refers to at least one member of **a class that is** the current instantiation or a non-dependent base class thereof. [*Note:* if no such member is found, and the current instantiation has any dependent base classes, then the *id-expression* is a member of an unknown specialization; see below. —*end note*]

   [*Example:* ... —*end example*]

   A name is a *dependent member of the current instantiation* if it is a member of the current instantiation which, when looked up, refers to at least one member of a class that is the current instantiation.

2. Change 17.7.2.1 [temp.dep.type] paragraph 5 as follows:

   A name is a *member of an unknown specialization* if it is

   - A *qualified-id* in which the *nested-name-specifier* names a dependent type that is not the current instantiation.
   - A *qualified-id* in which the *nested-name-specifier* refers to the current instantiation, the current instantiation has at least one dependent base class, and name lookup of the *qualified-id* does not find any member of **a class that is** the current instantiation or a non-dependent base class thereof.
   - An *id-expression* denoting the member in a class member access expression (8.2.5 [expr.ref]) in which either
     - the type of the object expression is the current instantiation, the current instantiation has at least one dependent base class, and name lookup of the *id-expression* does not find a member of **a class that is** the current instantiation or a non-dependent base class thereof; or
     - the type of the object expression is dependent and is not the current instantiation.

3. Change 17.7.2.1 [temp.dep.type] paragraph 8 as follows:

   A type is dependent if it is

   - ...
   - a nested class or enumeration that is a **dependent** member of the current instantiation,

- ...

4. Change 17.7.2.2 [temp.dep.expr] paragraph 3 as follows:

   An *id-expression* is type-dependent if it contains

   - ...

   or if it names a ~~static data~~ **dependent** member of the current instantiation that ~~has~~ **is a static data member of** type "array of unknown bound of `T`" for some `T` (17.6.1.3 [temp.static]). Expressions of the following forms...

5. Change 17.7.2.3 [temp.dep.constexpr] paragraph 2 as follows (assumes the base text is as modified by issue 1413):

   An *id-expression* is value-dependent if:

   - it is a name declared with a dependent type,
   - it is the name of a non-type template parameter,
   - it names a member of an unknown specialization,
   - it names a static data member ~~of the current instantiation~~ that is **a dependent member of the current instantiation and is** not initialized in a *member-declarator*,
   - it names a static member function that is a **dependent** member of the current instantiation, or
   - it is a constant with literal type and is initialized with an expression that is value-dependent.

   Expressions of the following form...

6. Change 17.7.2.3 [temp.dep.constexpr] paragraph 5 as follows (assumes the base text is as modified by issue 1413):

   An expression of the form `&`*qualified-id* where the *qualified-id*~~'s nested-name-specifier~~ names **a dependent member of** the current instantiation is value-dependent.

---

903. Value-dependent integral null pointer constants

Section: 17.7.2.3 [temp.dep.constexpr] **Status:** CD3 **Submitter:** Doug Gregor **Date:** 22 May, 2009

[Moved to DR status at the April, 2013 meeting.]

Consider the following example:

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

template <int N> void g() {
  f(N);
}

int main() {
  g<0>();
  g<1>();
}
```

The call to `f` in `g` is not type-dependent, so the overload resolution must be done at definition time rather than at instantiation time. As a result, both of the calls to `g` will result in calls to `f(...)`, i.e., `N` will not be a null pointer constant, even if the value of `N` is 0.

It would be most consistent to adopt a rule that a value-dependent expression can never be a null pointer constant, even in cases like

```
template <int N> void g() {
  int* p = N;
}
```

This would always be ill-formed, even when `N` is 0.

John Spicer: It's clear that this treatment is required for overload resolution, but it seems too expansive given that there are other cases in which the value of a template parameter can affect the validity of the program, and an implementation is forbidden to issue a diagnostic on a template definition unless there are no possible valid specializations.

Notes from the July, 2009 meeting:

There was a strong consensus among the CWG that only the literal `0` should be considered a null pointer constant, not any arbitrary zero-valued constant expression as is currently specified.

Proposed resolution (October, 2012):

1. Change 7.11 [conv.ptr] paragraph 1 as follows:

   A *null pointer constant* is an ~~integral constant expression (8.20 [expr.const]) prvalue of integer type that evaluates to~~ **integer literal (5.13.2 [lex.icon]) with value** zero or a prvalue of type `std::nullptr_t`. A null pointer constant can be converted...

2. Change 8.20 [expr.const] paragraph 3 as follows:

> ...[*Note:* Such expressions may be used as array bounds (11.3.4 [dcl.array], 8.3.4 [expr.new]), as bit-field lengths (12.2.4 [class.bit]), as enumerator initializers if the underlying type is not fixed (10.2 [dcl.enum]), ~~as null pointer constants (7.11 [conv.ptr]),~~ and as alignments (10.6.2 [dcl.align]). —*end note*]...

3. Change 11.6 [dcl.init] paragraph 5 as follows:

> To *zero-initialize* an object or reference of type `T` means:
>
> - if `T` is a scalar type (6.9 [basic.types]), the object is ~~set to the value `0` (zero), taken as an integral constant expression, converted~~ **initialized to the value obtained by converting integer literal `0` (zero)** to `T`; [*Footnote:* As specified in 7.11 [conv.ptr], converting an ~~integral constant expression~~ **integer literal** whose value is 0 to a pointer type results in a null pointer value. —*end footnote*]
> - ...

4. Change 17.3.2 [temp.arg.nontype] paragraph 5 as follows:

- ...
- for a non-type *template-parameter* of type pointer to object, qualification conversions (7.5 [conv.qual]) and the array-to-pointer conversion (7.2 [conv.array]) are applied; if the *template-argument* is of type `std::nullptr_t`, the null pointer conversion (7.11 [conv.ptr]) is applied. [*Note:* In particular, neither the null pointer conversion for a zero-valued ~~integral constant expression~~ **integer literal** (7.11 [conv.ptr]) nor the derived-to-base conversion (7.11 [conv.ptr]) are applied. Although 0 is...
- ...

5. Change 18.3 [except.handle] paragraph 3 as follows:

> ...[*Note:* A *throw-expression* whose operand is an ~~integral constant expression of integer type that evaluates to~~ **integer literal with value** zero does not match a handler of pointer or pointer to member type. —*end note*]. [*Example:* ...

6. Add a new section to C.2 [diff.cpp03] as follows:

> **C.2.x Clause 4: standard conversions** **[diff.cpp03.conv]**
>
> **7.11 [conv.ptr]**
> **Change: Only literals are integer null pointer constants**
> **Rationale: Removing surprising interactions with templates and constant expressions**
> **Effect on original feature: Valid C++ 2003 code may fail to compile or produce different results in this International Standard, as the following example illustrates:**

```
void f(void *);  // #1
void f(...);     // #2
template<int N> void g() {
    f(0*N);      // calls #2; used to call #1
}
```

Additional note (January, 2013):

Concerns were raised at the Portland (October, 2012) meeting that the value `false` has been used in existing code as a null pointer constant, and such code would be broken by this change. This issue has been returned to "review" status to allow discussion of whether to accommodate such code or not.

1413. Missing cases of value-dependency

Section: 17.7.2.3 [temp.dep.constexpr] **Status:** CD3 **Submitter:** Richard Smith **Date:** 2011-11-09

[Moved to DR at the April, 2013 meeting.]

The list of cases in 17.7.2.3 [temp.dep.constexpr] paragraph 2 in which an *identifier* is value-dependent should also include:

- an entity with reference type and is initialized with an expression that is value-dependent
- a member function or a static data member of the current instantiation

Proposed resolution (October, 2012):

1. Change 17.7.2.3 [temp.dep.constexpr] paragraph 2 as follows and move the text following the bulleted list into a new paragraph:

> An ~~*identifier*~~ ***id-expression*** is value-dependent if ~~it is~~:
>
> - **it is** a name declared with a dependent type,
> - **it is** the name of a non-type template parameter,

- **it names a member of an unknown specialization,**
- **it names a static data member of the current instantiation that is not initialized in a *member-declarator*,**
- **it names a static member function that is a member of the current instantiation, or**
- **it is** a constant with literal type and is initialized with an expression that is value-dependent.

Expressions of the following form...

2. Change 17.7.2.3 [temp.dep.constexpr] paragraph 5 as follows:

~~An *id-expression* is value-dependent if it names a member of an unknown specialization.~~ **An expression of the form `&`*qualified-id* where the *qualified-id*'s *nested-name-specifier* names the current instantiation is value-dependent.**

1532. Explicit instantiation and member templates

Section: 17.8.2 [temp.explicit] **Status:** CD3 **Submitter:** Johannes Schaub **Date:** 2012-08-04

[Moved to DR at the April, 2013 meeting.]

According to 17.8.2 [temp.explicit] paragraph 8,

> An explicit instantiation that names a class template specialization is also an explicit instantiation of the same kind (declaration or definition) of each of its members (not including members inherited from base classes) that has not been previously explicitly specialized in the translation unit containing the explicit instantiation, except as described below.

This could be read as an indication that member class templates and member function templates are instantiated (as templates) when their containing class template is instantiated. For example,

```
template<typename T> struct A {
  template<typename U> void f() {
    T t;
    t.f();
  }
};

template struct A<int>;
```

In this view, the result would be a member function template definition in class `A<int>` equivalent to

```
template<typename U> void f() {
  int t;
  t.f();
}
```

Such a template could never be validly instantiated and thus would presumably fall under the rule in 17.7 [temp.res] paragraph 8,

> If no valid specialization can be generated for a template, and that template is not instantiated, the template is ill-formed, no diagnostic required.

The wording of 17.8.2 [temp.explicit] paragraph 1 appears not to allow member templates to be instantiated as templates, however, mentioning only a "member template specialization" as a possibility:

> A class, a function or member template specialization can be explicitly instantiated from its template. A member function, member class or static data member of a class template can be explicitly instantiated from the member definition associated with its class template.

This appears to be a contradiction, and although a diagnostic for a member template such as the example above would be helpful, most or all current implementations do not do so. Either the wording of paragraph 1 should be changed to allow explicit instantiation of a member template as a template, analogous to explicitly specializing a member template as a template, or paragraph 8 should be clarified to exclude member templates from the members explicitly instantiated when the containing class template is explicitly instantiated.

Proposed resolution (October, 2012):

Change 17.8.2 [temp.explicit] paragraph 8 as follows:

> An explicit instantiation that names a class template specialization is also an explicit instantiation of the same kind (declaration or definition) of each of its members (not including members inherited from base classes **and members that are templates**) that has not been previously explicitly specialized in the translation unit containing the explicit instantiation, except as described below. [*Note:* In addition, it will typically be an explicit instantiation of certain implementation-dependent data about the class. —*end note*]

1227. Mixing immediate and non-immediate contexts in deduction failure

Section: 17.9.2 [temp.deduct] **Status:** CD3 **Submitter:** Daniel Krügler **Date:** 2010-11-27

[Moved to DR at the October, 2012 meeting.]

Consider the following example:

```
template <int> struct X {
  typedef int type;
};
template <class T> struct Y { };
template <class T> struct Z {
  static int const value = Y<T>::value;
};

template <class T> typename X<Y<T>::value + Z<T>::value>::type f(T);
int f(...);

int main() {
  sizeof f(0);
}
```

The problem here is that there is a combination of an invalid expression in the immediate context (`Y<T>::value`) and in the non-immediate context (within `Z<T>` when evaluating `Z<T>::value`). The Standard does not appear to state clearly whether this program is well-formed (because the error in the immediate context causes deduction failure) or ill-formed (because of the error in the non-immediate context).

Notes from the March, 2011 meeting:

Some members expressed a desire to allow implementations latitude in whether examples like this should be deduction failure or a diagnosable error, just as the order of evaluation of arithmetic operands is largely unconstrained. Others felt that specifying something like a depth-first left-to-right traversal of the expression or declaration would be better. Another possibility suggested was to enforce ordering only at comma operators. No consensus was achieved.

CWG agreed that the arguments should be processed in left-to-right order. Some popular existing code (e.g., Boost) depends on this ordering.

Proposed resolution (February, 2012):

Change 17.9.2 [temp.deduct] paragraph 7 as follows:

> The substitution occurs in all types and expressions that are used in the function type and in template parameter declarations. The expressions include not only constant expressions such as those that appear in array bounds or as nontype template arguments but also general expressions (i.e., non-constant expressions) inside `sizeof`, `decltype`, and other contexts that allow non-constant expressions. **The substitution proceeds in lexical order and stops when a condition that causes deduction to fail is encountered.** [*Note:* The equivalent substitution in exception specifications is done only when the function is instantiated, at which point a program is ill-formed if the substitution results in an invalid type or expression. — *end note*] **[*Example:*
>
> ```
> template <class T> struct A { using X = typename T::X; };
> template <class T> typename T::X f(typename A<T>::X);
> template <class T> void f(...) { }
> template <class T> auto g(typename A<T>::X) -> typename T::X;
> template <class T> void g(...) { }
>
> void h() {
>   f<int>(0); // OK, substituting return type causes deduction to fail
>   g<int>(0); // error, substituting parameter type instantiates A<int>
> }
> ```
>
> — *end example*]

1262. Default template arguments and deduction failure

Section: 17.9.2 [temp.deduct] **Status:** CD3 **Submitter:** Nikolay Ivchenkov **Date:** 2011-03-16

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

Proposed resolution (August, 2011):

Change 17.9.2 [temp.deduct] paragraph 5 as follows:

> The resulting substituted and adjusted function type is used as the type of the function template for template argument deduction. If a template argument has not been deduced~~, its default template argument, if any, is used~~ **and its corresponding template parameter has a default argument, the template argument is determined by substituting the template arguments determined for preceding template parameters into the default argument. If the substitution results in an invalid type, as described above, type deduction fails**. [*Example:*...

1330. Delayed instantiation of `noexcept` specifiers

Section: 17.9.2 [temp.deduct] **Status:** CD3 **Submitter:** Jason Merrill **Date:** 2011-06-05

[Moved to DR at the April, 2013 meeting.]

See also issues 595 and 287.

The use of `noexcept` specifiers can cause instantiation of classes and functions that are not actually needed in the program, just to be able to complete the declaration. The actual value of the expression in the *noexcept-specification* is only needed if the function is defined (i.e., instantiated) or called, so it would significantly reduce the number of instantiations (and avoid certain kinds of errors when the value is currently required before a class is complete) if *exception-specification*s were treated like default arguments and only instantiated when they are actually needed.

Proposed resolution (February, 2012):

1. Change 17.6 [temp.decls] paragraph 2 as follows:

   For purposes of name lookup and instantiation, default arguments **and *exception-specification*s** of function templates and default arguments **and *exception-specification*s** of member functions of class templates are considered definitions; each default argument **or *exception-specification*** is a separate definition which is unrelated to the function template definition or to any other default arguments **or *exception-specification*s**.

2. Change 17.7 [temp.res] paragraph 11 as follows:

   [*Note:* For purposes of name lookup, default arguments **and *exception-specification*s** of function templates and default arguments **and *exception-specification*s** of member functions of class templates are considered definitions (14.5). —*end note*]

3. Add a new paragraph following 17.7.4.1 [temp.point] paragraph 2:

   If a function template or member function of a class template is called in a way which uses the definition of a default argument of that function template or member function, the point of instantiation of the default argument is the point of instantiation of the function template or member function specialization.

   For an *exception-specification* of a function template specialization or specialization of a member function of a class template, if the *exception-specification* is implicitly instantiated because it is needed by another template specialization and the context that requires it depends on a template parameter, the point of instantiation of the *exception-specification* is the point of instantiation of the specialization that requires it. Otherwise, the point of instantiation for such an *exception-specification* immediately follows the namespace scope declaration or definition that requires the *exception-specification*.

4. Change 17.8.1 [temp.inst] paragraph 1 as follows:

   ...The implicit instantiation of a class template specialization causes the implicit instantiation of the declarations, but not of the definitions**,** ~~or~~ default arguments, **or *exception-specification*s,** of the class member functions, member classes...

5. Add a new paragraph following 17.8.1 [temp.inst] paragraph 13:

   Each default argument is instantiated independently. [*Example:* ... —*end example*]

   If the *exception-specification* of a specialization of a function template or member function of a class template has not yet been instantiated, but is needed (e.g., to instantiate the function definition, to evaluate a *noexcept-expression* (8.3.7 [expr.unary.noexcept]), or to compare against the *exception-specification* of another declaration), the dependent names are looked up, the semantic constraints are checked, and the instantiation of any template used in the *exception-specification* is done as if it were being done as part of instantiating the declaration of the specialization. An *exception-specification* is not instantiated in order to calculate the *exception-specification* of a defaulted function in a derived class until the *exception-specification* of the derived member function becomes necessary.

6. Change the note 17.9.2 [temp.deduct] paragraph 7 as follows:

   ...[*Note:* The equivalent substitution in exception specifications is done only when the ~~function~~ ***exception-specification*** is instantiated, at which point a program is ill-formed if the substitution results in an invalid type or expression. —*end note*]

7. Add a new paragraph following 18.4 [except.spec] paragraph 15:

   A deallocation function (6.7.4.2 [basic.stc.dynamic.deallocation]) with no explicit *exception-specification* is treated as if it were specified with `noexcept(true)`.

   The *exception-specification* of a function template specialization is not instantiated along with the function declaration; it is instantiated when needed (17.8.1 [temp.inst]). The *exception-specification* of an implicitly-declared special member function is also evaluated as needed. [*Note:* Therefore, an implicit declaration of a member function of a derived class does not require the *exception-specification* of a base member function to be instantiated. —*end note*]

Notes from the February, 2012 meeting:

There should be a specific definition of when an *exception-specification* is needed and must thus be instantiated.

Additional discussion (September, 2012):

Daveed Vandevoorde brought up two additional points. First, the rule should be crafted so that an example like the following is ill-formed:

```
template<class T> T f() noexcept(sizeof(T) < 4);

int main() {
  decltype(f<void>()) *p;
}
```

Even though the *exception-specification* is not needed here, it should be instantiated (because of the unevaluated reference to `f`) in order to catch the ill-formed `sizeof(T)`.

In addition, the proposed change creates an asymmetry between class templates and ordinary classes:

```
struct S {
  void f() noexcept(sizeof(g()) < 8); // Invalid forward reference.
  static int g();
};
```

but

```
template<typename> struct X {
    void f() noexcept(sizeof(g()) < 8); // Okay.
    static int g();
};
```

If the proposed change is adopted, the rules for *exception-specification*s in ordinary classes should be revised to make the parallel usage well-formed.

Proposed resolution (October, 2012):

1. Change 6.3.7 [basic.scope.class] paragraph 1 #1 as follows:

   The potential scope of a name declared in a class consists not only of the declarative region following the name's point of declaration, but also of all function bodies, default arguments, ***exception-specification*s,** and *brace-or-equal-initializer*s of non-static data members in that class (including such things in nested classes).

2. Change 6.4.1 [basic.lookup.unqual] paragraph 7 as follows:

   A name used in the definition of a class `X` outside of a member function body, default argument, ***exception-specification,*** *brace-or-equal-initializer* of a non-static data member, or nested class definition[29] shall be declared in one of the following ways:...

3. Change 6.4.1 [basic.lookup.unqual] paragraph 8 as follows:

   For the members of a class `X`, a name used in a member function body, in a default argument, **in an *exception-specification*,** in the *brace-or-equal-initializer* of a non-static data member (12.2 [class.mem]), or in the definition of a class member outside of the definition of `X`, following the member's *declarator-id*[31] , shall be declared in one of the following ways:...

4. Change 12.2 [class.mem] paragraph 2 as follows:

   A class is considered a completely-defined object type (6.9 [basic.types]) (or complete type) at the closing } of the *class-specifier*. Within the class *member-specification*, the class is regarded as complete within function bodies, default arguments, ***exception-specification*s,** and *brace-or-equal-initializer*s for non-static data members (including such things in nested classes). Otherwise it is regarded as incomplete within its own class *member-specification*.

5. Change 17.6 [temp.decls] paragraph 2 as follows:

   For purposes of name lookup and instantiation, default arguments **and *exception-specification*s** of function templates and default arguments **and *exception-specification*s** of member functions of class templates are considered definitions; each default argument **or *exception-specification*** is a separate definition which is unrelated to the function template definition or to any other default arguments **or *exception-specification*s**.

6. Change 17.7 [temp.res] paragraph 11 as follows:

   [*Note:* For purposes of name lookup, default arguments **and *exception-specification*s** of function templates and default arguments **and *exception-specification*s** of member functions of class templates are considered definitions (17.6 [temp.decls]). —*end note*]

7. Add the following as a new paragraph after 17.7.4.1 [temp.point] paragraph 2:

   If a function template or member function of a class template is called in a way which uses the definition of a default argument of that function template or member function, the point of instantiation of the default argument is the point of instantiation of the function template or member function specialization.

   For an *exception-specification* of a function template specialization or specialization of a member function of a class template, if the *exception-specification* is implicitly instantiated because it is needed by another template specialization and the context that requires it depends on a template parameter, the point of instantiation of the *exception-specification* is the point of instantiation of the specialization that requires it. Otherwise, the point of instantiation for such an *exception-specification* immediately follows the namespace scope declaration or definition that requires the *exception-specification*.

8. Change 17.8.1 [temp.inst] paragraph 1 as follows:

Unless a class template specialization has been explicitly instantiated (17.8.2 [temp.explicit]) or explicitly specialized (17.8.3 [temp.expl.spec]), the class template specialization is implicitly instantiated when the specialization is referenced in a context that requires a completely-defined object type or when the completeness of the class type affects the semantics of the program. The implicit instantiation of a class template specialization causes the implicit instantiation of the declarations, but not of the definitions**,** ~~or~~ default arguments, **or *exception-specification*s** of the class member functions, member classes, scoped member enumerations, static data members and member templates; and it causes the implicit instantiation of the definitions of unscoped member enumerations and member anonymous unions. However, for the purpose...

9. Insert the following as a new paragraph immediately preceding 17.8.1 [temp.inst] paragraph 14:

The *exception-specification* of a function template specialization is not instantiated along with the function declaration; it is instantiated when needed (18.4 [except.spec]). If such an *exception-specification* is needed but has not yet been instantiated, the dependent names are looked up, the semantics constraints are checked, and the instantiation of any template used in the *exception-specification* is done as if it were being done as part of instantiating the declaration of the specialization at that point.

[*Note:* 17.7.4.1 [temp.point] defines the point of instantiation of a template specialization. —*end note*]

10. Change 17.9.2 [temp.deduct] paragraph 7 as follows:

The substitution occurs in all types and expressions that are used in the function type and in template parameter declarations. The expressions include not only constant expressions such as those that appear in array bounds or as nontype template arguments but also general expressions (i.e., non-constant expressions) inside sizeof, decltype, and other contexts that allow non-constant expressions. [*Note:* The equivalent substitution in exception specifications is done only when the ~~function~~ ***exception-specification*** is instantiated, at which point a program is ill-formed if the substitution results in an invalid type or expression. —*end note*]

11. Change 18.4 [except.spec] paragraph 2 as follows:

...A type denoted in an *exception-specification* shall not denote an incomplete type ~~other than a class currently being defined~~. A type denoted in an *exception-specification* shall not denote a pointer or reference to an incomplete type, other than *cv* `void*` ~~or a pointer or reference to a class currently being defined~~. A type *cv* `T`, "array of `T`", or "function returning `T`" denoted in an *exception-specification* is adjusted to type `T`, "pointer to `T`", or "pointer to function returning `T`" respectively.

12. Add the following as a new paragraph following 18.4 [except.spec] paragraph 15:

A deallocation function (6.7.4.2 [basic.stc.dynamic.deallocation]) with no explicit *exception-specification* is treated as if it were specified with `noexcept(true)`.

An *exception-specification* is considered to be *needed* when:

- **in an expression, the function is the unique lookup result or the selected member of a set of overloaded functions (6.4 [basic.lookup], 16.3 [over.match], 16.4 [over.over]);**
- **the function is odr-used (6.2 [basic.def.odr]) or, if it appears in an unevaluated operand, would be odr-used if the expression were potentially-evaluated;**
- **the *exception-specification* is compared to that of another declaration (e.g. an explicit specialization or an overriding virtual function);**
- **the function is defined; or**
- **the *exception-specification* is needed for a defaulted special member function that calls the function. [*Note:* A defaulted declaration does not require the *exception-specification* of a base member function to be evaluated until the implicit *exception-specification* of the derived function is needed, but an explicit *exception-specification* needs the implicit *exception-specification* to compare against. —*end note*]**

The *exception-specification* of a defaulted special member function is evaluated as described above only when needed; similarly, the *exception-specification* of a specialization of a function template or member function of a class template is instantiated only when needed.

[Note: this resolution reverses the decision in issue 1308.]

1462. Deduction failure vs "ill-formed, no diagnostic required"

Section: 17.9.2 [temp.deduct] **Status:** CD3 **Submitter:** John Spicer **Date:** 2012-02-08

[Moved to DR at the April, 2013 meeting.]

The relationship between errors that render a program ill-formed but for which no diagnostic is required and things that cause deduction failure is not clearly specified. Presumably failures that need not be diagnosed cannot be the basis for SFINAE, lest different implementations perform deduction differently depending on how thoroughly they handle such cases. This should be spelled out explicitly.

Proposed resolution (October, 2012):

Change 17.9.2 [temp.deduct] paragraph 8 as follows:

> If a substitution results in an invalid type or expression, type deduction fails. An invalid type or expression is one that would be ill-formed**, with a diagnostic required,** if written using the substituted arguments. [*Note:* **If no diagnostic is required, the program is still ill-formed.** Access checking is done as part of the substitution process. —*end note*] Only invalid types and expressions...

1388. Missing non-deduced context following a function parameter pack

Section: 17.9.2.1 [temp.deduct.call] **Status:** CD3 **Submitter:** James Widman **Date:** 2011-09-02

[Moved to DR at the October, 2012 meeting.]

Presumably 17.9.2.5 [temp.deduct.type] paragraph 5 should include a bullet for a function parameter or function parameter pack that follows a function parameter pack.

Proposed resolution (February, 2012):

Change 17.9.2.1 [temp.deduct.call] paragraph 1 as follows:

> ~~...For a function parameter pack that does not occur at the end of the *parameter-declaration-list*, the type of the parameter pack is a non-deduced context.~~ **When a function parameter pack appears in a non-deduced context (17.9.2.5 [temp.deduct.type]), the type of that parameter pack is never deduced.** [*Example:*
>
> ```
> template<class ... Types> void f(Types& ...);
> template<class T1, class ... Types> void g(T1, Types ...);
> template<class T1, class ... Types> void g1(Types ..., T1);
>
> void h(int x, float& y) {
>   const int z = x;
>   f(x, y, z);                 // Types is deduced to int, float, const int
>   g(x, y, z);                 // T1 is deduced to int; Types is deduced to float, int
>   g1(x, y, z);                    // error: Types is not deduced
>   g1<int, int, int>(x, y, z);  // OK, no deduction occurs
> }
> ```
>
> —*end example*]

This resolution also resolves issue 1399.

1399. Deduction with multiple function parameter packs

Section: 17.9.2.1 [temp.deduct.call] **Status:** CD3 **Submitter:** Jason Merrill **Date:** 2011-09-29

[Moved to DR at the October, 2012 meeting.]

Consider:

```
template <class... T>
void f(T..., int, T...) { }

int main() {
   f(0);          // OK
   f<int>(0,0,0); // OK
   f(0,0,0);      // error
}
```

It seems clear that the third call is ill-formed because by the time we get to the second function parameter pack we've already assumed that `T` is empty, so deducing anything for `T` would be nonsensical. But I don't think this is expressed anywhere in the standard.

One way to handle this would be to say that a template parameter pack is not deducible if it is used in a function parameter pack not at the end of the parameter list.

Proposed resolution (February, 2012):

This issue is resolved by the resolution of issue 1388.

1372. Cross-references incorrect in conversion function template argument deduction

Section: 17.9.2.3 [temp.deduct.conv] **Status:** CD3 **Submitter:** Michael Wong **Date:** 2011-08-15

[Moved to DR at the October, 2012 meeting.]

According to 17.9.2.3 [temp.deduct.conv] paragraph 1,

> Template argument deduction is done by comparing the return type of the conversion function template (call it P; see 11.6 [dcl.init], 16.3.1.5 [over.match.conv], and 16.3.1.6 [over.match.ref] for the determination of that type) with the type that is required as the result of the conversion (call it A) as described in 17.9.2.5 [temp.deduct.type].

It would seem that the cross-references should apply to the determination of the type "required as the result of the conversion" (i.e., A) instead of the return type of the conversion function.

Proposed resolution (February, 2012):

Change 17.9.2.3 [temp.deduct.conv] paragraph 1 as follows:

> Template argument deduction is done by comparing the return type of the conversion function template (call it P~~; see 11.6 [dcl.init], 16.3.1.5 [over.match.conv], and 16.3.1.6 [over.match.ref] for the determination of that type~~) with the type that is required as the result of the conversion (call it A**; see 11.6 [dcl.init], 16.3.1.5 [over.match.conv], and 16.3.1.6 [over.match.ref] for the determination of that type**) as described in 17.9.2.5 [temp.deduct.type].

1387. Missing non-deduced context for `decltype`

Section: 17.9.2.5 [temp.deduct.type] **Status:** CD3 **Submitter:** James Widman **Date:** 2011-09-02

[Moved to DR at the October, 2012 meeting.]

Presumably 17.9.2.5 [temp.deduct.type] paragraph 5 should include a bullet for a *decltype-specifier* whose *expression* references a template parameter.

Proposed resolution (February, 2012):

Change 17.9.2.5 [temp.deduct.type] paragraph 5 as follows:

> The non-deduced contexts are:
>
> - The *nested-name-specifier* of a type that was specified using a *qualified-id*.
> - **The *expression* of a *decltype-specifier*.**
> - A non-type template argument...

1431. Exceptions from other than *throw-expression*s

Section: 18 [except] **Status:** CD3 **Submitter:** Nikolay Ivchenkov **Date:** 2011-12-16

[Moved to DR at the October, 2012 meeting.]

There are a number of places in the Standard that appear to assume that exceptions are only thrown by *throw-expression*s. Various other constructs, such as `dynamic_cast`s, `typeid`, *new-expression*s, etc., can also throw exceptions, so a more general term should be coined and applied in place of *throw-expression* wherever necessary.

Proposed resolution (February, 2012):

1. Change 6.7.4.1 [basic.stc.dynamic.allocation] paragraph 3 as follows:

   > ...Any other allocation function that fails to allocate storage shall indicate failure only by throwing an exception **(18.1 [except.throw])** of a type that would match a handler (18.3 [except.handle]) of type `std::bad_alloc` (21.6.3.1 [bad.alloc]).

2. Change 6.7.4.1 [basic.stc.dynamic.allocation] paragraph 4 as follows:

   > ...[*Note:* In particular, a global allocation function is not called to allocate storage for objects with static storage duration (6.7.1 [basic.stc.static]), for objects or references with thread storage duration (6.7.2 [basic.stc.thread]), for objects of type `std::type_info` (8.2.8 [expr.typeid]), or for ~~the copy of an object thrown by a throw expression~~ **an exception object** (18.1 [except.throw]). —*end note*]

3. Change 8.2.7 [expr.dynamic.cast] paragraph 9 as follows:

   > The value of a failed cast to pointer type is the null pointer value of the required result type. A failed cast to reference type throws **an exception (18.1 [except.throw]) of a type that would match a handler (18.3 [except.handle]) of type** `std::bad_cast` (21.7.3 [bad.cast]).

4. Change 8.2.8 [expr.typeid] paragraph 2 as follows:

   > ...If the glvalue expression is obtained by applying the unary * operator to a pointer[68] and the pointer is a null pointer value (7.11 [conv.ptr]), the `typeid` expression throws ~~the~~ **an exception (18.1 [except.throw]) of a type that would match**

a handler of type `std::bad_typeid` exception (21.7.4 [bad.typeid]).

5. Change 8.3.4 [expr.new] paragraph 7 as follows:

   When the value of the *expression* in a *noptr-new-declarator* is zero, the allocation function is called to allocate an array with no elements. If the value of that *expression* is less than zero or such that the size of the allocated object would exceed the implementation-defined limit, or if the *new-initializer* is a *braced-init-list* for which the number of *initializer-clause*s exceeds the number of elements to initialize, no storage is obtained and the *new-expression* ~~terminates by throwing~~ **throws** an exception **(18.1 [except.throw])** of a type that would match a handler (18.3 [except.handle]) of type `std::bad_array_new_length` (21.6.3.2 [new.badlength]).

6. Change 18 [except] paragraph 1 as follows:

   ...A handler will be invoked only by ~~a *throw-expression* invoked~~ **throwing an exception** in code executed in the handler's try block or in functions called from the handler's try block...

7. Change 18 [except] paragraph 2 as follows:

   A *try-block* is a *statement* (Clause 9 [stmt.stmt]). A *throw-expression* is of type `void`. ~~Code that executes a *throw-expression* is said to "throw an exception;" code that subsequently gets control is called a "handler."~~ [*Note:*...

8. Change 18.1 [except.throw] paragraph 1 as follows:

   Throwing an exception transfers control to a handler. **[*Note:* An exception can be thrown from one of the following contexts: *throw-expression* (see below), allocation functions (6.7.4.1 [basic.stc.dynamic.allocation]),** `dynamic_cast` **(8.2.7 [expr.dynamic.cast]),** `typeid` **(8.2.8 [expr.typeid]), *new-expression* (8.3.4 [expr.new]), and standard library functions (20.4.1.4 [structure.specifications]). —*end note*]** An object is passed and the type of that object determines which handlers can catch it. [*Example:*...

9. Change 18.1 [except.throw] paragraph 3 as follows:

   ~~A *throw-expression*~~ **Throwing an exception** copy-initializes (11.6 [dcl.init]**, 15.8 [class.copy]**) a temporary object, called the *exception object*~~, the type of which is determined by removing any top-level *cv-qualifier*s from the static type of the operand of `throw` and adjusting the type from "array of `T`" or "function returning `T`" to "pointer to `T`" or "pointer to function returning `T`", respectively~~. The temporary is an lvalue and is used to initialize the variable named in the matching *handler* (18.3 [except.handle]). If the type of the exception object would be an incomplete type or a pointer to an incomplete type other than (possibly cv-qualified) `void` the program is ill-formed. ~~Except for these restrictions and the restrictions on type matching mentioned in 18.3 [except.handle], the operand of `throw` is treated exactly as a function argument in a call (8.2.2 [expr.call]) or the operand of a return statement.~~ **Evaluating a *throw-expression* with an operand throws an exception; the type of the exception object is determined by removing any top-level *cv-qualifier*s from the static type of the operand and adjusting the type from "array of `T`" or "function returning `T`" to "pointer to `T`" or "pointer to function returning `T`," respectively.**

10. Change 18.1 [except.throw] paragraph 4 as follows:

    ...[*Note:* ~~an~~ **a thrown** exception ~~thrown by a *throw-expression*~~ does not propagate to other threads unless caught, stored, and rethrown using appropriate library functions; see 21.8.6 [propagation] and 33.6 [futures]. —*end note*]

11. Change 18.1 [except.throw] paragraph 8 as follows:

    A *throw-expression* with no operand rethrows the currently handled exception (18.3 [except.handle]). The exception is reactivated with the existing ~~temporary~~ **exception object**; no new ~~temporary~~ exception object is created. The exception is no longer considered to be caught; therefore, the value of `std::uncaught_exception()` will again be `true`. [*Example:*...

12. Change 18.2 [except.ctor] paragraph 1 as follows:

    As control passes from ~~a *throw-expression*~~ **the point where an exception is thrown** to a handler, destructors are invoked for all automatic objects constructed since the try block was entered...

13. Change 18.2 [except.ctor] paragraph 3 as follows:

    The process of calling destructors for automatic objects constructed on the path from a try block to ~~a *throw-expression*~~ **the point where an exception is thrown** is called "*stack unwinding*." If a destructor...

14. Change 18.3 [except.handle] paragraph 17 as follows:

    When the handler declares ~~a non-constant~~ **an** object, any changes to that object will not affect the ~~temporary object that was initialized by execution of the *throw-expression*~~ **exception object**. When the handler declares a reference to ~~a non-constant~~ **an** object, any changes to the referenced object are changes to the ~~temporary object initialized when the *throw-expression* was executed~~ **exception object** and will have effect should that object be rethrown.

15. Change 21.8.4.4 [terminate] paragraph 1 as follows:

    Remarks: Called by the implementation when exception handling must be abandoned for any of several reasons (18.5.1 [except.terminate]), in effect immediately after ~~evaluating the *throw-expression* (21.8.4.1 [terminate.handler])~~ **throwing the exception**. May also be called directly by the program.

1503. Exceptions during copy to exception object

Section: 18.1 [except.throw] **Status:** CD3 **Submitter:** Daniel Krügler **Date:** 2012-05-11

[Moved to DR at the April, 2013 meeting.]

According to 18.1 [except.throw] paragraph 7,

> If the exception handling mechanism, after completing evaluation of the expression to be thrown but before the exception is caught, calls a function that exits via an exception, `std::terminate` is called (18.5.1 [except.terminate]).

This wording was overlooked in the resolution for issue 475 and should be changed, along with the following example, to indicate that `std::terminate` will be called for an uncaught exception only after initialization of the exception object is complete.

Proposed resolution (August, 2012):

Change 18.1 [except.throw] paragraph 7 as follows:

> If the exception handling mechanism, after completing ~~evaluation of the expression to be thrown~~ **the initialization of the exception object** but before the ~~exception is caught~~ **activation of a handler for the exception**, calls a function that exits via an exception, `std::terminate` is called (18.5.1 [except.terminate]). [*Example:*
>
> ```
> struct C {
>   C() { }
>   C(const C&) { throw 0; }
>     if (std::uncaught_exception()) {
>       throw 0;     // throw during copy to handler's exception-declaration object (18.3 [except.handle])
>     }
>   }
> };
>
> int main() {
>   try {
>     throw C();   // calls std::terminate() if construction of the handler's
>                  // exception-declaration object is not elided (15.8 [class.copy])
>   } catch(C) { }
> }
> ```
>
> —*end example*]

---

388. Catching base*& from a throw of derived*

Section: 18.3 [except.handle] **Status:** CD3 **Submitter:** John Spicer **Date:** 28 Oct 2002

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

I have a question about exception handling with respect to derived to base conversions of pointers caught by reference.

What should the result of this program be?

```
struct S             {};
struct SS : public S {};

int main()
{
      SS ss;
      int result = 0;
      try
      {
              throw &ss; // throw object has type SS*
                         // (pointer to derived class)
      }
      catch (S*& rs) // (reference to pointer to base class)
      {
              result = 1;
      }
      catch (...)
      {
              result = 2;
      }
      return result;
}
```

The wording of 18.3 [except.handle] paragraph 3 would seem to say that the catch of S*& does not match and so the catch ... would be taken.

All of the compilers I tried (EDG, g++, Sun, and Microsoft) used the catch of S*& though.

What do we think is the desired behavior for such cases?

My initial reaction is that this is a bug in all of these compilers, but the fact that they all do the same thing gives me pause.

On a related front, if the handler changes the parameter using the reference, what is caught by a subsequent handler?

```
extern "C" int printf(const char *, ...);
struct S             {};
```

```
struct SS : public S {};
SS ss;

int f()
{
	try
	{
		throw &ss;
	}
	catch (S*& rs) // (reference to pointer to base class)
	{
		rs = 0;
		throw;
	}
	catch (...)
	{
	}
	return 0;
}

int main()
{
	try { f(); }
	catch (S*& rs) {
		printf("rs=%p, &ss=%p\n", rs, &ss);
	}
}
```

EDG, g++, and Sun all catch the original (unmodified) value. Microsoft catches the modified value. In some sense the EDG/g++/Sun behavior makes sense because the later catch could catch the derived class instead of the base class, which would be difficult to do if you let the catch clause update the value to be used by a subsequent catch.

But on this non-pointer case, all of the compilers later catch the modified value:

```
extern "C" int printf(const char *, ...);
int f()
{
	try
	{
		throw 1;
	}
	catch (int& i)
	{
		i = 0;
		throw;
	}
	catch (...)
	{
	}
	return 0;
}

int main()
{
	try { f(); }
	catch (int& i) {
		printf("i=%p\n", i);
	}
}
```

To summarize:

1. Should "base*const&" be able to catch a "derived*"? The current standard seems to say "no" but parallels to how calls work, and existing practice, suggest that the answer should be "yes".
2. Should "base*&" be able to catch a "derived*". Again, the standard seems seems to say "no". Parallels to how calls work still suggest "no", but existing practice suggests "yes".
3. If either of the above is "yes", what happens if you modify the pointer referred to by the reference. This requires a cast to remove const for case #2.
4. On a related front, if you catch "derived*&" when a "derived*" is thrown, what happens if you modify the pointer referred to by the reference? EDG/g++/Sun still don't modify the underlying value that would be caught by a rethrow in this case. This case seems like it should be the same as the "int&" example above, but is not on the three compilers mentioned.

(See also issue 729.)

Notes from the October, 2009 meeting:

The consensus of the CWG was that it should not be possible to catch a pointer to a derived class using a reference to a base class pointer, and that a handler that takes a reference to non-const pointer should allow the pointer to be modified by the handler.

Proposed resolution (March, 2010):

Change 18.3 [except.handle] paragraph 3 as follows:

> A *handler* is a match for an exception object of type `E` if
>
> - The *handler* is of type *cv* `T` or *cv* `T&` and `E` and `T` are the same type (ignoring the top-level *cv-qualifier*s), or
> - the *handler* is of type *cv* `T` or *cv* `T&` and `T` is an unambiguous public base class of `E`, or
> - the *handler* is of type *cv1* `T` ~~** cv2*~~ **or** `const T&` **where** `T` **is a pointer type** and `E` is a pointer type that can be converted to ~~the type of the handler~~ `T` by either or both of

- a standard pointer conversion (7.11 [conv.ptr]) not involving conversions to pointers to private or protected or ambiguous classes
- a qualification conversion

- the *handler* is **of type *cv*** `T` **or** `const T&` **where** `T` **is** a pointer or pointer to member type and `E` is `std::nullptr_t`.

(This resolution also resolves issue 729.)

Notes from the March, 2011 meeting:

This resolution would require an ABI change and was thus deferred for further consideration.

---

729. Qualification conversions and handlers of reference-to-pointer type

Section: 18.3 [except.handle] **Status:** CD3 **Submitter:** John Spicer **Date:** 6 October, 2008

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

Given the following example:

```
int f() {
    try { /* ... */ }
    catch(const int*&) {
        return 1;
    }
    catch(int*&) {
        return 2;
    }
    return 3;
}
```

can `f()` return `2`? That is, does an `int*` exception object match a `const int*&` handler?

According to 18.3 [except.handle] paragraph 3, it does not:

A *handler* is a match for an exception object of type `E` if

- The *handler* is of type *cv* `T` or *cv* `T&` and `E` and `T` are the same type (ignoring the top-level *cv-qualifier*s), or
- the *handler* is of type *cv* `T` or *cv* `T&` and `T` is an unambiguous public base class of `E`, or
- the handler is of type *cv1* `T*` *cv2* and `E` is a pointer type that can be converted to the type of the handler by either or both of
  - a standard pointer conversion (7.11 [conv.ptr]) not involving conversions to pointers to private or protected or ambiguous classes
  - a qualification conversion
- the *handler* is a pointer or pointer to member type and `E` is `std::nullptr_t`.

Only the third bullet allows qualification conversions, but only the first bullet applies to a *handler* of reference-to-pointer type. This is consistent with how other reference bindings work; for example, the following is ill-formed:

```
int* p;
const int*& r = p;
```

(The consistency is not complete; the reference binding would be permitted if `r` had type `const int* const &`, but a handler of that type would still not match an `int*` exception object.)

However, implementation practice seems to be in the other direction; both EDG and g++ do match an `int*` with a `const int*&`, and the Microsoft compiler issues an error for the presumed hidden handler in the code above. Should the Standard be changed to reflect existing practice?

(See also issue 388.)

Notes from the October, 2009 meeting:

The CWG agreed that matching the exception object with a handler should, to the extent possible, mimic ordinary reference binding in cases like this.

Proposed resolution (February, 2010):

This issue is resolved by the resolution of issue 388.

1267. Rvalue reference types in *exception-specification*s

Section: 18.4 [except.spec] **Status:** CD3 **Submitter:** Michael Wong **Date:** 2011-03-21

[Moved to DR at the October, 2012 meeting.]

The types that may appear in an *exception-specification* (18.4 [except.spec] paragraph 2) include rvalue reference types, although they are excluded as handler types (18.3 [except.handle] paragraph 1). This appears to have been an oversight.

Proposed resolution (February, 2012):

Change 18.4 [except.spec] paragraph 2 as follows:

> ...A type denoted in an *exception-specification* shall not denote an incomplete type **or an rvalue reference type**. A type denoted in an *exception-specification* shall not denote a pointer or reference...

1282. Underspecified destructor *exception-specification*

Section: 18.4 [except.spec] **Status:** CD3 **Submitter:** Daniel Krügler **Date:** 2011-03-28

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

It is not clear whether the unexpected handler will be invoked in the following example:

```
#include <iostream>
#include <exception>

struct A { ~A() throw() { } };
struct B { ~B() noexcept { } };
struct C : A, B { ~C() { throw 0; } };

void unexpected_observer() {
 std::cerr << "unexpected called" << std::endl;
 std::terminate();
}

int main() {
 std::set_unexpected(unexpected_observer);
 C c;
}
```

The problem is 18.4 [except.spec] paragraph 14 only says that the *exception-specification* of `C::~C` "shall allow no exceptions," which could mean either `throw()` or `noexcept(true)`.

Proposed resolution (August, 2011):

Change 18.4 [except.spec] paragraph 14 as follows:

> An implicitly declared special member function (Clause 15 [special]) ~~shall have~~ **has** an *exception-specification*. If `f` is an implicitly declared default constructor, copy constructor, move constructor, destructor, copy assignment operator, or move assignment operator, its implicit *exception-specification* specifies the *type-id* `T` if and only if `T` is allowed by the *exception-specification* of a function directly invoked by `f`'s implicit definition; `f` ~~shall allow~~ **allows** all exceptions if any function it directly invokes allows all exceptions, and `f` ~~shall allow no exceptions~~ **has the *exception-specification*** `noexcept(true)` if every function it directly invokes allows no exceptions. [*Example:*
>
> ```
> struct A {
>   A();
>   A(const A&) throw();
>   A(A&&) throw();
>   ~A() throw(X);
> };
> struct B {
>   B() throw();
>   B(const B&) throw();
>   B(B&&) throw(Y);
>   ~B() throw(Y);
> };
> struct D : public A, public B {
>     // Implicit declaration of D::D();
>     // Implicit declaration of D::D(const D&) ~~throw()~~ noexcept(true);
>     // Implicit declaration of D::D(D&&) throw(Y);
>     // Implicit declaration of D::~D() throw(X, Y);
> };
> ```
>
> Furthermore, if...

1381. Implicitly-declared special member functions and default `nothrow`

Section: 18.4 [except.spec] **Status:** CD3 **Submitter:** David Svoboda **Date:** 2011-08-26

[Moved to DR at the October, 2012 meeting.]

According to 18.4 [except.spec] paragraph 14,

> An implicitly declared special member function (Clause 15 [special]) shall have an *exception-specification*. If `f` is an implicitly declared default constructor, copy constructor, move constructor, destructor, copy assignment operator, or move assignment operator, its implicit *exception-specification* specifies the *type-id* `T` if and only if `T` is allowed by the *exception-specification* of a function directly invoked by `f`'s implicit definition; `f` shall allow all exceptions if any function it directly invokes allows all exceptions, and `f` shall allow no exceptions if every function it directly invokes allows no exceptions.

It would be clearer if this description made explicit the intent that special member functions that invoke no other functions are to allow no exceptions.

Proposed resolution (February, 2012):

Change 18.4 [except.spec] paragraph 14 as follows:

> ...and `f` has the *exception-specification* `noexcept(true)` if every function it directly invokes allows no exceptions. **[*Note:* It follows that `f` has the *exception-specification* `noexcept(true)` if it invokes no other functions. —*end note*]** [*Example:*

```
struct A {
  A();
  A(const A&) throw();
  A(A&&) throw();
  ~A() throw(X);
};
struct B {
  B() throw();
  B(const B&) ~~throw()~~ = default;  // Declaration of B::B(const B&) noexcept(true)
  B(B&&) throw(Y);
  ~B() throw(Y);
};
...
```

1370. *identifier-list* cannot contain ellipsis

Section: 19.3 [cpp.replace] **Status:** CD3 **Submitter:** Nikolay Ivchenkov **Date:** 2011-08-14

[Moved to DR at the October, 2012 meeting.]

19.3 [cpp.replace] paragraph 12 says,

> If there is a `...` in the *identifier-list* in the macro definition...

However, an *identifier-list* cannot contain an ellipsis according to the grammar in 19 [cpp] paragraph 1.

Proposed resolution (February, 2012):

Change 19.3 [cpp.replace] paragraph 12 as follows:

> If there is a `...` ~~in the identifier-list~~ **immediately preceding the** `)` in the **function-like** macro definition, then the trailing arguments, including any separating comma preprocessing tokens, are merged to form a single item: the ~~variable-arguments~~ ***variable arguments***. The number of arguments so combined is such that, following merger, the number of arguments is one more than the number of parameters in the macro definition (excluding the `...`).

1329. Recursive deduction substitutions

Section: B [implimits] **Status:** CD3 **Submitter:** Jason Merrill **Date:** 2011-06-02

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

It is not clear whether the implementation limit on recursion in template instantiation applies only to instantiation itself or also to recursion that occurs during template argument deduction.

Proposed resolution (August, 2011):

Change B [implimits] as follows:

- Recursively nested template instantiations**, including substitution during template argument deduction (17.9.2 [temp.deduct])** [1 024].

1251. C compatibility: casting to unqualified `void*`

Section: C.1.3 [diff.conv] **Status:** CD3 **Submitter:** Johannes Schaub **Date:** 2011-03-04

[Voted into the WP at the February, 2012 meeting; moved to DR at the October, 2012 meeting.]

The incompatibility described appears not to exist.

Proposed resolution (August, 2011):

Delete the second entry in C.1.3 [diff.conv], i.e., the one headed by

> **Change:** Only pointers to non-const and non-volatile objects may be implicitly converted to `void*`

2114. Missing description of incompatibility from aggregate NSDMIs

Section: C.3.5 [diff.cpp11.dcl.decl] **Status:** CD3 **Submitter:** Ville Voutilainen **Date:** 2015-04-14

The following example illustrates an incompatibility between C++11 and C++14:

```
struct S {
  int m = 1;
};        // C++11: S is non-aggregate
          // C++14: S is AGGREGATE
struct X {
    operator int();
    operator S();
};

int main() {
  X a{};
  S b{a};  // C++11: valid, copy constr S(a.operator S()) is called here
           // C++14: valid, aggregate initialization { a.operator int() }

  printf("%d\n", b.m);
}
```

This should be documented in Annex C [diff].

Notes from the May, 2015 meeting:

This will be handled editorially; the status has been set to "review" to check that the editorial change has been made.

---

223. The meaning of deprecation

Section: D [depr] **Status:** CD3 **Submitter:** Mike Miller **Date:** 19 Apr 2000

[Moved to DR at the April, 2013 meeting.]

During the discussion of issues 167 and 174, it became apparent that there was no consensus on the meaning of deprecation. Some thought that deprecating a feature reflected an intent to remove it from the language. Others viewed it more as an encouragement to programmers not to use certain constructs, even though they might be supported in perpetuity.

There is a formal-sounding definition of deprecation in Annex D [depr] paragraph 2:

> deprecated is defined as: Normative for the current edition of the Standard, but not guaranteed to be part of the Standard in future revisions.

However, this definition would appear to say that any non-deprecated feature *is* "guaranteed to be part of the Standard in future revisions." It's not clear that that implication was intended, so this definition may need to be amended.

This issue is intended to provide an avenue for discussing and resolving those questions, after which the original issues may be reopened if that is deemed desirable.

Proposed resolution (August, 2012):

Change D [depr] paragraph 2 as follows:

> These are deprecated features, where *deprecated* is defined as: Normative for the current edition of the Standard, but ~~not guaranteed to be part of the Standard in~~ **having been identified as a candidate for removal from** future revisions.

Issues with "CD4" Status

1882. Reserved names without library use

Section: _N4140_.17.6.4.3.2 [global.names] **Status:** CD4 **Submitter:** Mike Miller **Date:** 2014-02-26

[Moved to DR at the November, 2014 meeting.]

The section of the Standard reserving names that begin with two underscores or an underscore and a capital letter, _N4140_.17.6.4.3.2 [global.names], applies only to "programs that use the facilities of the C++ standard library" (20.5.4.1 [constraints.overview]). However, implementations rely on this restriction for mangling, even when no standard library facilities are used. Should this requirement be moved to the core language section?

(There is a related issue with user-defined literal suffixes, 20.5.4.3.5 [usrlit.suffix]. However, these are already mentioned normatively in the core language section, so it could be argued that the question of library usage does not apply.)

Proposed resolution (October, 2014):

1. Change 5.10 [lex.name] paragraph 3 as follows:

   In addition, some identifiers are reserved for use by C++ implementations ~~and standard libraries (_N4140_.17.6.4.3.2 [global.names])~~ and shall not be used otherwise; no diagnostic is required.

   - **Each identifier that contains a double understore `__` or begins with an underscore followed by an uppercase letter is reserved to the implementation for any use.**
   - **Each identifier that begins with an underscore is reserved to the implementation for use as a name in the global namespace.**

2. Change the footnote in 11.4.1 [dcl.fct.def.general] paragraph 8 as follows:

   [*Footnote:* Implementations are permitted to provide additional predefined variables with names that are reserved to the implementation (~~_N4140_.17.6.4.3.2 [global.names]~~ **5.10 [lex.name]**). If a predefined variable is not odr-used (6.2 [basic.def.odr]), its string value need not be present in the program image. —*end footnote*]

3. Change the example in 16.5.8 [over.literal] paragraph 8 as follows:

```
double operator"" _Bq(double);  // OK: does not use the reserved ~~name~~ identifier _Bq (~~_N4140_.17.6.4.3.2 [global.names]~~ 5.10 [lex.name])
double operator"" _Bq(double); // uses the reserved ~~name~~ identifier _Bq (~~_N4140_.17.6.4.3.2 [global.names]~~ 5.10 [lex.name])
```

4. Delete _N4140_.17.6.4.3.2 [global.names]:

   ~~Certain sets of names and function signatures are always reserved to the implementation:~~

   - ~~Each name that contains a double underscore `__` or begins with an underscore followed by an uppercase letter (5.11 [lex.key]) is reserved to the implementation for any use.~~
   - ~~Each name that begins with an underscore is reserved to the implementation for use as a name in the global namespace.~~

1573. Inherited constructor characteristics

Section: _N4527_.12.9 [class.inhctor] **Status:** CD4 **Submitter:** Jason Merrill **Date:** 2012-10-15

[Moved to DR at the November, 2014 meeting.]

Issue 1350 clarified that the *exception-specification* for an inheriting constructor is determined like defaulted functions, but we should also say something similar for `deleted`, and perhaps `constexpr`.

Also, the description of the semantics of inheriting constructors don't seem to allow for C-style variadic functions, so the text should be clearer that such constructors are only inherited without their ellipsis.

Proposed resolution (February, 2014):

1. Change _N4527_.12.9 [class.inhctor] paragraph 1 as follows:

   A *using-declaration* (10.3.3 [namespace.udecl]) that names a constructor implicitly declares a set of *inheriting constructors*. The *candidate set of inherited constructors* from the class `X` named in the *using-declaration* consists of actual constructors and notional constructors that result from the transformation of defaulted parameters **and ellipsis parameter specifications** as follows:

   - ~~all non-template constructors~~ **for each non-template constructor** of `X`, **the constructor that results from omitting any ellipsis parameter specification,** and
   - for each non-template constructor of `X` that has at least one parameter with a default argument, the set of constructors that results from omitting any ellipsis parameter specification and successively omitting parameters with a default argument from the end of the parameter-type-list, and
   - ~~all constructor templates~~ **for each constructor template** of `X`, **the constructor template that results from omitting any ellipsis parameter specification,** and

- for each constructor template of `X` that has at least one parameter with a default argument, the set of constructor templates that results from omitting any ellipsis parameter specification and successively omitting parameters with a default argument from the end of the parameter-type-list.

2. Change _N4527_.12.9 [class.inhctor] paragraph 2 as follows:

   The *constructor characteristics* of a constructor or constructor template are

   - the template parameter list (17.1 [temp.param]), if any,
   - the ~~*parameter-type-list*~~ **parameter-type-list** (11.3.5 [dcl.fct]), **and**
   - absence or presence of `explicit` (15.3.1 [class.conv.ctor])~~, and~~**.**
   - ~~absence or presence of `constexpr` (10.1.5 [dcl.constexpr]).~~

3. Change _N4527_.12.9 [class.inhctor] paragraph 4 as follows:

   A constructor so declared has the same access as the corresponding constructor in `X`. **It is `constexpr` if the user-written constructor (see below) would satisfy the requirements of a `constexpr` constructor (10.1.5 [dcl.constexpr]).** It is deleted if the corresponding constructor in `X` is deleted (~~11.4 [dcl.fct.def]~~ **11.4.3 [dcl.fct.def.delete]**) **or if a defaulted default constructor (15.1 [class.ctor]) would be deleted, except that the construction of the direct base class `X` is not considered in the determination**. An inheriting constructor shall not be explicitly instantiated (17.8.2 [temp.explicit]) or explicitly specialized (17.8.3 [temp.expl.spec]).

1645. Identical inheriting constructors via default arguments

Section: _N4527_.12.9 [class.inhctor] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2013-03-18

[Adopted at the October, 2015 meeting as P0136R1.]

For an example like

```
struct A {
  constexpr A(int, float = 0);
  explicit A(int, int = 0);
  A(int, int, int = 0) = delete;
};

struct B : A {
  using A::A;
};
```

it is not clear from _N4527_.12.9 [class.inhctor] what should happen: is `B::B(int) constexpr` and/or `explicit`? Is `B::B(int, int) explicit` and/or deleted? Although the rationale given in the note in paragraph 7,

> If two *using-declaration*s declare inheriting constructors with the same signatures, the program is ill-formed (12.2 [class.mem], 16.1 [over.load]), because an implicitly-declared constructor introduced by the first *using-declaration* is not a user-declared constructor and thus does not preclude another declaration of a constructor with the same signature by a subsequent *using-declaration*.

might be thought to apply, paragraph 1 talks about a *set* of candidate constructors based on their parameter types, so presumably such a set would contain only a single declaration of `A::A(int)` and one for `A::A(int, int)`. The constructor characteristics of that declaration, however, are not specified.

One possibility might be to declare such a constructor, resulting from the transformation of more than one base class constrctor, to be deleted, so there would be an error only if it were used.

Notes from the April, 2013 meeting:

CWG agreed with the direction of defining such constructors as deleted.

Additional note, June, 2014:

See issue 1941 for an alternative approach to this problem.

1715. Access and inherited constructor templates

Section: _N4527_.12.9 [class.inhctor] **Status:** CD4 **Submitter:** Daveed Vandevoorde **Date:** 2013-07-16

[Adopted at the October, 2015 meeting as P0136R1.]

Consider the following example:

```
template<class T> struct S {
private:
```

```
  typedef int X;
  friend struct B;
};

struct B {
  template<class T> B(T, typename T::X);
};

struct D: B {
  using B::B;
};

S<int> s;
B b(s, 2); // Okay, thanks to friendship.
D d(s, 2); // Error: friendship is not inherited.
```

My understanding is that the construction of `d` fails because `typename T::X` expands to `S<int>::X` in this case, and that is not accessible from `D`.

However, I'm not sure that makes sense from a usability perspective. The user of `D` just wanted to be able to wrap class `B`, and the fact that friendship was granted to `B` to enable its constructor parameter seems like just an implementation detail that `D` shouldn't have to cope with.

Would it perhaps be better to suspend access checking during the instantiation of inheriting member function template declarations (not definitions), since real access problems (e.g., the selection of a private constructor) would presumably be revealed when doing the full instantiation?

Proposed resolution (February, 2014):

Change _N4527_.12.9 [class.inhctor] paragraph 4 as follows:

> A constructor so declared has the same access as the corresponding constructor in `X`. It is deleted if the corresponding constructor in `X` is deleted (~~11.4 [dcl.fct.def]~~ **11.4.3 [dcl.fct.def.delete]**). **While performing template argument substitution (17.9.2 [temp.deduct]) for constructor templates so declared, name lookup, overload resolution, and access checking are performed in the context of the corresponding constructor template of `X`. [*Example:***
>
> ```
> struct B {
>   template<class T> B(T, typename T::Q);
> };
>
> class S {
>   using Q = int;
>   template<class T>
>   friend B::B(T, typename T::Q);
> };
>
> struct D : B {
>   using B::B;
> };
>
> B b(S(), 1); // OK: B::B is a friend of S
> D d(S(), 2); // OK: access control is in the context of B::B
> ```
>
> ***—end example*]** An inheriting constructor shall not be explicitly instantiated (17.8.2 [temp.explicit]) or explicitly specialized (17.8.3 [temp.expl.spec]).

Additional note (June, 2014):

This issue is being returned to "review" status in light of a suggestion for an alternative approach to the problem; see issue 1941.

1736. Inheriting constructor templates in a local class

Section: _N4527_.12.9 [class.inhctor] **Status:** CD4 **Submitter:** Daveed Vandevoorde **Date:** 2013-08-13

[Adopted at the October, 2015 meeting as P0136R1.]

A local class cannot, according to 17.6.2 [temp.mem] paragraph 2, have member templates. Presumably, then, an example like the following is ill-formed:

```
struct S {
  template<class T> S(T) {
    struct L: S {
      using S::S;
    };
  }
};
```

It is accepted by current implementations, however. Does something need to be said about this case in _N4527_.12.9 [class.inhctor], either to explicitly allow or forbid it, or is the restriction in 17.6.2 [temp.mem] sufficient?

1941. SFINAE and inherited constructor default arguments

Section: _N4527_.12.9 [class.inhctor] **Status:** CD4 **Submitter:** David Krauss **Date:** 2014-06-12

[Adopted at the October, 2015 meeting as P0136R1.]

Default arguments are a common mechanism for applying SFINAE to constructors. However, default arguments are not carried over when base class constructors are inherited; instead, an overload set of constructors with various numbers of arguments is created in the derived class. This seems problematic.

One possibility would be to change the mechanism for how constructors are inherited; a *using-declaration* might actually introduce the base class constructors into the derived class, as other *using-declaration*s do, and if a base class constructor were selected, the remaining derived class members would be default-initialized.

This approach would also address issues 1645, as duplicated constructors would simply fail during overload resolution, and 1715, since there would be no synthesized constructors for which access checking would be needed.

The effect of such an approach on ABIs, including mangling, would need to be considered.

See also issue 1959.

1959. Inadvertently inherited copy constructor

Section: _N4527_.12.9 [class.inhctor] **Status:** CD4 **Submitter:** David Krauss **Date:** 2014-06-30

[Adopted at the October, 2015 meeting as P0136R1.]

Consider the following example:

```
struct a {
  a() = default;
  a( a const & ) { std::cout << "copy\n"; }
  template< typename t >
  a( t ) { std::cout << "convert\n"; }
};

struct b : a {
  using a::a;
};

a x;
b y = x;
```

The copy constructor is invoked by the inherited constructor template, making it effectively inherited, contrary to the intent of _N4527_.12.9 [class.inhctor] paragraph 3. `std::function` is affected by this issue.

A kernel of a resolution might be to inherit the copy and move constructors as deleted. Then they will be more specialized than any template, and the user won't get conversion-from-base behavior unless they explicitly declare it. However, reference binding in overload resolution is a potential gap. Something like `b::b( a & ) = delete` with a non-const parameter would not add safety if it's not chosen.

See also issue 1941.

1991. Inheriting constructors vs default arguments

Section: _N4527_.12.9 [class.inhctor] **Status:** CD4 **Submitter:** Hubert Tong **Date:** 2014-08-27

[Adopted at the October, 2015 meeting as P0136R1.]

The creation of inheriting constructors does not, but should, consider the default arguments of constructors in the inheriting class. For example,

```
struct A {
  A(int, int);
};

struct B : A {
  using A::A;
  B(int, int, int = 0); // does not suppress creation of B(int, int) from A(int, int)
};
```

1929. `template` keyword following namespace *nested-name-specifier*

Section: _N4567_.5.1.1 [expr.prim.general] **Status:** CD4 **Submitter:** David Krauss **Date:** 2014-05-15

[Moved to DR at the May, 2015 meeting.]

It is not clear whether the `template` keyword should be accepted in an example like

```
template<typename> struct s {};
::template s<void> q; // innocuous disambiguation?
```

Although it is accepted by the grammar, the verbiage in _N4567_.5.1.1 [expr.prim.general] paragraph 10 does not mention the possibility, while the preceding paragraph dealing with class qualification calls it out explicitly.

Notes from the June, 2014 meeting:

CWG agreed that this usage should be accepted.

Proposed resolution (November, 2014):

Change _N4567_.5.1.1 [expr.prim.general] paragraph 10 as follows (the base wording is as modified by issue 1887):

> The *nested-name-specifier* `::` names the global namespace. A *nested-name-specifier* that names a namespace (10.3 [basic.namespace]), **optionally followed by the keyword** `template` **(17.2 [temp.names]), and then** followed by the name of a member of that namespace (or the name of a member of a namespace made visible by a *using-directive*), is a *qualified-id*...

2124. Signature of constructor template

Section: 3.23 [defns.signature.member.templ] **Status:** CD4 **Submitter:** Hubert Tong **Date:** 2015-05-05

[Adopted at the February, 2016 meeting.]

According to 3.23 [defns.signature.member.templ], the signature of a class member function template includes:

> name, parameter type list (11.3.5 [dcl.fct]), class of which the function is a member, *cv*-qualifiers (if any), *ref-qualifier* (if any), return type, and template parameter list

However, a constructor template does not have a return type. This may be relevant to friend declaration matching.

Proposed resolution (October, 2015):

Change 3.23 [defns.signature.member.templ] as follows:

> **signature**
> <class member function template> name, parameter type list (11.3.5 [dcl.fct]), class of which the function is a member, *cv-qualifier*s (if any), *ref-qualifier* (if any), return type **(if any)**, and template parameter list

---

1949. "sequenced after" instead of "sequenced before"

Section: 4.6 [intro.execution] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-06-18

[Moved to DR at the October, 2015 meeting.]

The term "sequenced after" is used in both the core and library clauses instead of the more-correct "sequenced before."

Proposed resolution (May, 2015):

1. Change 4.6 [intro.execution] paragraph 13 as follows:

> *Sequenced before* is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread (4.7 [intro.multithread]), which induces a partial order among those evaluations. Given any two evaluations *A* and *B*, if *A* is sequenced before *B* **(or, equivalently, *B* is *sequenced after A*)**, then the execution of *A* shall precede the execution of *B*. If *A* is not sequenced before *B*...

2. Change 4.7 [intro.multithread] paragraph 14 as follows:

> An evaluation *A* *happens before* an evaluation *B* **(or, equivalently, *B happens after A*)** if:...

3. Change 4.6 [intro.execution] paragraph 15 as follows:

> ...~~Every evaluation in the calling function (including other function calls) that is not otherwise specifically sequenced before or after the execution of the body of the called function is indeterminately sequenced with respect to the execution of the called function~~ **For each function invocation *F*, for every evaluation *A* that occurs within *F* and every evaluation *B* that does not occur within *F* but is evaluated on the same thread and as part of the same signal handler (if any), either *A* is sequenced before *B* or *B* is sequenced before *A*.**[9] **[*Note:* if *A* and *B* would not otherwise be sequenced then they are indeterminately sequenced. —*end note*]** Several contexts...

4. Change 6.6.2 [basic.start.static] paragraph 4 as follows:

It is implementation-defined whether the dynamic initialization of a non-local variable with static storage duration ~~is done~~ **happens** before the first statement of `main`. If the initialization is deferred to ~~some point in time~~ **happen** after the first statement of `main`, it ~~shall occur~~ **happens** before the first odr-use (6.2 [basic.def.odr]) of any function or variable...

5. Change 6.6.2 [basic.start.static] paragraph 5 as follows:

> It is implementation-defined whether the dynamic initialization of a non-local variable with static or thread storage duration is ~~done~~ **sequenced** before the first statement of the initial function of the thread. If the initialization is deferred to some point in time **sequenced** after the first statement of the initial function of the thread, it ~~shall occur~~ **is sequenced** before the first odr-use (6.2 [basic.def.odr]) of any variable with thread storage duration defined in the same translation unit as the variable to be initialized.

6. Change 9.5.3 [stmt.for] paragraph 1 as follows:

> ...[*Note:* Thus the first statement specifies initialization for the loop; the condition (9.4 [stmt.select]) specifies a test, ~~made~~ **sequenced** before each iteration, such that the loop is exited when the condition becomes `false`; the expression often specifies incrementing that is ~~done~~ **sequenced** after each iteration. —*end note*]

7. Add the following as a new paragraph at the end of 18 [except]:

> **In this section, "before" and "after" refer to the "sequenced before" relation (4.6 [intro.execution]).**

2146. Scalar object vs memory location in definition of "unsequenced"

Section: 4.6 [intro.execution] **Status:** CD4 **Submitter:** Jens Maurer **Date:** 2015-06-22

[Adopted at the February, 2016 meeting.]

According to 4.6 [intro.execution] paragraph 15,

> If a side effect on a scalar object is unsequenced relative to either another side effect on the same scalar object or a value computation using the value of the same scalar object, and they are not potentially concurrent (4.7 [intro.multithread]), the behavior is undefined.

Should this refer to "memory location," which also encompasses contiguous bit-fields, as the definition of data races in 4.7 [intro.multithread] does? For example,

```
struct S {
  int x : 4;
  int y : 4;
  int z : 4;
};

void f(int, int, int);
int g(int, S&);

int main(int argc, char ** argv) {
  S s = { argc, argc+1, argc+2 };
  f(++s.x, g(++s.y, s), ++s.z);
}
```

Proposed resolution (February, 2016):

Change 4.6 [intro.execution] paragraph 15 as follows:

> ...If a side effect on a ~~scalar object~~ **memory location (4.4 [intro.memory])** is unsequenced relative to either another side effect on the same ~~scalar object~~ **memory location** or a value computation using the value of **any object in** the same ~~scalar object~~ **memory location**, and they are not potentially concurrent (4.7 [intro.multithread]), the behavior is undefined. [*Note:* The next section...

1999. Representation of source characters as universal-character-names

Section: 5.2 [lex.phases] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-09-09

[Moved to DR at the May, 2015 meeting.]

According to 5.2 [lex.phases] paragraph 1, first phase,

> Any source file character not in the basic source character set (5.3 [lex.charset]) is replaced by the universal-character-name that designates that character. (An implementation may use any internal encoding, so long as an actual extended character encountered in the source file, and the same extended character expressed in the source file as a universal-character-name (i.e., using the `\uXXXX` notation), are handled equivalently except where this replacement is reverted in a raw string literal.)

This wording is obviously not intended to exclude the use of characters with code points larger than `0xffff`, but the reference to "the `\uXXXX` notation" might suggest that the `\Uxxxxxxxx` form is not allowed.

Proposed resolution (April, 2015):

Change 5.2 [lex.phases] paragraph 1 number 1 as follows:

> ...(An implementation may use any internal encoding, so long as an actual extended character encountered in the source file, and the same extended character expressed in the source file as a universal-character-name (~~i.e.~~ **e.g.**, using the `\uXXXX` notation), are handled equivalently except where this replacement is reverted in a raw string literal.)

1796. Is all-bits-zero for null characters a meaningful requirement?

Section: 5.3 [lex.charset] **Status:** CD4 **Submitter:** Tony van Eerd **Date:** 2013-10-02

[Moved to DR at the November, 2014 meeting.]

According to 5.3 [lex.charset] paragraph 3,

> The *basic execution character set* and the *basic execution wide-character set* shall each contain all the members of the basic source character set, plus control characters representing alert, backspace, and carriage return, plus a *null character* (respectively, *null wide character*), whose representation has all zero bits.

It is not clear that a portable program can examine the bits of the representation; instead, it would appear to be limited to examining the bits of the numbers corresponding to the value representation (6.9.1 [basic.fundamental] paragraph 1). It might be more appropriate to require that the null character value compare equal to 0 or `'\0'` rather than specifying the bit pattern of the representation.

There is a similar issue for the definition of shift, bitwise `and`, and bitwise `or` operators: are those specifications constraints on the bit pattern of the representation or on the values resulting from the interpretation of those patterns as numbers?

Proposed resolution (February, 2014):

Change 5.3 [lex.charset] paragraph 3 as follows:

> The *basic execution character set* and the *basic execution wide-character set* shall each contain all the members of the basic source character set, plus control characters representing alert, backspace, and carriage return, plus a *null character* (respectively, *null wide character*), whose ~~representation has all zero bits~~ **value is 0**. For each basic execution character set...

2000. *header-name* outside `#include` directive

Section: 5.4 [lex.pptoken] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-09-09

[Moved to DR at the October, 2015 meeting.]

The "max munch" rule could be read to require the characters `<int>` in `vector<int>` to be parsed as a *header-name* rather than as three distinct tokens. 5.8 [lex.header] paragraph 1 says,

> Header name preprocessing tokens shall only appear within a `#include` preprocessing directive (19.2 [cpp.include]).

However, that is not sufficiently clear that *header-name*s are only to be recognized in that context.

Proposed resolution (May, 2015):

1. Change 5.4 [lex.pptoken] bullet 3.3 as follows:
   - Otherwise, the next preprocessing token is the longest sequence of characters that could constitute a preprocessing token, even if that would cause further lexical analysis to fail**, except that a *header-name* (5.8 [lex.header]) is only formed within a `#include` directive (19.2 [cpp.include])**.
2. Change 5.8 [lex.header] paragraph 1 as follows:

   > **[*Note:* Header name preprocessing tokens ~~shall~~ only appear within a `#include` preprocessing directive (~~19.2 [cpp.include]~~ **see 5.4 [lex.pptoken]**). ***—end note*]** The sequences in both forms...

1963. Implementation-defined identifier characters

Section: 5.10 [lex.name] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-07-07

[Moved to DR at the May, 2015 meeting.]

The grammar in 5.10 [lex.name] includes the production,

> *identifier-nondigit:*

nondigit
universal-character-name
other implementation-defined characters

The rule for "other implementation-defined characters" is a holdover from before the introduction of universal-character-names, intended to allow the use of non-ASCII national characters in identifiers. However, since all characters outside the basic source character set are now conceptually mapped to universal-character-names in translation phase 1, there is no longer a need for such a provision and it should be removed.

Proposed resolution (April, 2015):

Change the grammar in 5.10 [lex.name] as follows:

identifier-nondigit:
nondigit
universal-character-name
~~other implementation-defined characters~~

1802. `char16_t` **string literals and surrogate pairs**

Section: 5.13.5 [lex.string] **Status:** CD4 **Submitter:** Jeffrey Yasskin **Date:** 2013-10-30

[Moved to DR at the November, 2014 meeting.]

The intent of `char16_t` string literals, as evident from 5.13.5 [lex.string] paragraph 9, is that they be encoded in UTF-16, that is, including surrogate pairs to represent code points outside the basic multi-lingual plane:

> A single *c-char* may produce more than one `char16_t` character in the form of surrogate pairs.

Paragraph 15, however, is inconsistent with this approach, saying,

> Escape sequences and universal-character-names in non-raw string literals have the same meaning as in character literals (5.13.3 [lex.ccon]), except that the single quote ' is representable either by itself or by the escape sequence `\'`, and the double quote " shall be preceded by a `\`.

The reason is that code points outside the basic multi-lingual plane are ill-formed in `char16_t` character literals:

> A character literal that begins with the letter `u`, such as `u'y'`, is a character literal of type `char16_t`. The value of a `char16_t` literal containing a single *c-char* is equal to its ISO 10646 code point value, provided that the code point is representable with a single 16-bit code unit. (That is, provided it is a basic multi-lingual plane code point.) If the value is not representable within 16 bits, the program is ill-formed.

It should be clarified that this restriction does not apply to `char16_t` string literals.

Proposed resolution (February, 2014):

Change 5.13.5 [lex.string] paragraph 16 as follows:

> Escape sequences and universal-character-names in non-raw string literals have the same meaning as in character literals (5.13.3 [lex.ccon]), except that the single quote ' is representable either by itself or by the escape sequence `\'`, and the double quote " shall be preceded by a `\`**, and except that a universal-character-name in a** `char16_t` **string literal may yield a surrogate pair**. In a narrow string literal...

1810. Invalid *ud-suffix*es

Section: 5.13.8 [lex.ext] **Status:** CD4 **Submitter:** Gabriel Dos Reis **Date:** 2013-11-13

[Moved to DR at the November, 2014 meeting.]

In explaining the relationship between preprocessing tokens and tokens, 5.4 [lex.pptoken] paragraph 4 contains the following example:

> [*Example:* The program fragment `1Ex` is parsed as a preprocessing number token (one that is not a valid floating or integer literal token), even though a parse as the pair of preprocessing tokens `1` and `Ex` might produce a valid expression (for example, if `Ex` were a macro defined as `+1`).

This analysis does not take into account the addition of user-defined literals. In fact, `1Ex` matches the rule for a *user-defined-integer-literal*, which is then ill-formed because it uses a reserved *ud-suffix* (5.13.8 [lex.ext] paragraph 10), as well as (presumably) because of a lookup failure for a matching literal operator, raw literal operator, or literal operator template.

More generally, it might be preferable to eliminate the restriction on the use of a reserved *ud-suffix* and rely simply on the fact that it is ill-formed to declare a literal operator, raw literal operator, or literal operator template with a reserved literal suffix identifier (20.5.4.3.5 [usrlit.suffix], cf 16.5.8 [over.literal] paragraph 1).

Proposed resolution (June, 2014):

1. Change 5.4 [lex.pptoken] paragraph 4 as follows:

   [*Example:* The program fragment ~~1Ex~~ `0xe+foo` is parsed as a preprocessing number token (one that is not a valid floating or integer literal token), even though a parse as ~~the pair of~~ **three** preprocessing tokens `1`, `0xe`, `+`, and ~~Ex~~ `foo` might produce a valid expression (for example, if ~~Ex~~ `foo` were a macro defined as ~~-~~`1`). Similarly, the program fragment `1E1` is parsed as a preprocessing number (one that is a valid floating literal token), whether or not `E` is a macro name. —*end example*]

2. Delete 5.13.8 [lex.ext] paragraph 10:

   ~~Some identifiers appearing as *ud-suffix*es are reserved for future standardization (20.5.4.3.5 [usrlit.suffix]). A program containing such a *ud-suffix* is ill-formed, no diagnostic required.~~

3. Change 16.5.8 [over.literal] paragraph 1 as follows:

   The *string-literal* or *user-defined-string-literal* in a *literal-operator-id* shall have no *encoding-prefix* and shall contain no characters other than the implicit terminating '\0'. The *ud-suffix* of the *user-defined-string-literal* or the identifier in a *literal-operator-id* is called a *literal suffix identifier*. ~~[*Note:* some~~ **Some** literal suffix identifiers are reserved for future standardization; see 20.5.4.3.5 [usrlit.suffix]. ~~—*end note*]~~ **A declaration whose *literal-operator-id* uses such a literal suffix identifier is ill-formed; no diagnostic required.**

4. Change 20.5.4.3.5 [usrlit.suffix] paragraph 1 as follows:

   Literal suffix identifiers **(16.5.8 [over.literal])** that do not start with an underscore are reserved for future standardization.

Additional note, May, 2014:

It has been suggested that the change to 5.4 [lex.pptoken] paragraph 4 in the proposed resolution would be simpler and better if it did not venture into questions about user-defined literals but simply relied on a string that is a valid pp-number but not a valid floating-point number, as was the case before the introduction of user-defined literals, e.g., 1.2.3.4. The issue has been returned to "review" status for discussion of this suggestion.

1870. Contradictory wording about definitions vs explicit specialization/instantiation

Section: 6.1 [basic.def] **Status:** CD4 **Submitter:** Hubert Tong **Date:** 2014-02-15

[Moved to DR at the November, 2014 meeting.]

Sections 17.8.2 [temp.explicit] and 17.8.3 [temp.expl.spec] describe cases of explicit instantiation directives and explicit specializations, respectively, that are not definitions. However, the description in 6.1 [basic.def] does not include these distinctions, classifying all declarations other than those listed as definitions. These should be harmonized.

Proposed Resolution (July, 2014):

Change 6.1 [basic.def] paragraph 2 as follows:

> A declaration is a *definition* unless it... an *empty-declaration* (Clause 10 [dcl.dcl]), ~~or~~ a *using-directive* (10.3.4 [namespace.udir])**, an explicit instantiation declaration (17.8.2 [temp.explicit]), or an explicit specialization (17.8.3 [temp.expl.spec]) whose *declaration* is not a definition**.

1614. Address of pure virtual function vs odr-use

Section: 6.2 [basic.def.odr] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2013-01-31

[Moved to DR at the November, 2014 meeting.]

According to 6.2 [basic.def.odr] paragraph 3,

> A function whose name appears as a potentially-evaluated expression is odr-used if it is the unique lookup result or the selected member of a set of overloaded functions (6.4 [basic.lookup], 16.3 [over.match], 16.4 [over.over]), unless it is a pure virtual function and its name is not explicitly qualified.

In the following example, consequently, `S::f` is odr-used but not defined, and (because it is an undefined odr-used inline function) a diagnostic is required:

```
namespace {
  struct S {
    inline virtual void f() = 0;
  };
 void (S::*p) = &S::f;
}
```

However, `S::f` cannot be called through such a pointer-to-member, so forming a pointer-to-member should not cause a pure virtual function to be odr-used. There is implementation divergence on this point.

Proposed resolution (April, 2013):

Change 6.2 [basic.def.odr] paragraph 3 as follows:

> ...A virtual member function is odr-used if it is not pure. A function whose name appears as a potentially-evaluated expression is odr-used if it is the unique lookup result or the selected member of a set of overloaded functions (6.4 [basic.lookup], 16.3 [over.match], 16.4 [over.over]), unless it is a pure virtual function and **either** its name is not explicitly qualified **or the expression forms a pointer to member (5.3.1)**. [*Note:*...

1926. Potential results of subscript operator

Section: 6.2 [basic.def.odr] **Status:** CD4 **Submitter:** Marcel Wid **Date:** 2014-05-13

[Moved to DR at the May, 2015 meeting.]

The definition of the potential results of an expression in 6.2 [basic.def.odr] paragraph 2 do not, but should, include the subscript operator.

Proposed resolution (November, 2014):

Change 6.2 [basic.def.odr] paragraph 2 as follows:

> ...The set of *potential results* of an expression `e` is defined as follows:
>
> - If `e` is an *id-expression* (_N4567_.5.1.1 [expr.prim.general]), the set contains only `e`.
> - **If `e` is a subscripting operation (8.2.1 [expr.sub]) with an array operand, the set contains that operand.**
> - If `e` is a class member access...

2085. Invalid example of adding special member function via default argument

Section: 6.2 [basic.def.odr] **Status:** CD4 **Submitter:** Hubert Tong **Date:** 2015-02-13

[Adopted at the February, 2016 meeting.]

The example in 6.2 [basic.def.odr] bullet 6.6 reads,

```
//translation unit 1:
struct X {
  X(int);
  X(int, int);
};
X::X(int = 0) { }
class D: public X { };
D d2;     // X(int) called by D()

//translation unit 2:
struct X {
  X(int);
  X(int, int);
};
X::X(int = 0, int = 0) { }
class D: public X { };   // X(int, int) called by D();
                         // D()'s implicit definition
                         // violates the ODR
```

Creating a special member function via default arguments added in an out-of-class definition, as is done here, is no longer permitted, so at a minimum the example should be removed. It is not clear whether there remain any cases to which the normative wording of bullet 6.6 would apply:

- if `D` is a class with an implicitly-declared constructor (15.1 [class.ctor]), it is as if the constructor was implicitly defined in every translation unit where it is odr-used, and the implicit definition in every translation unit shall call the same constructor for a base class or a class member of `D`.

If not, the entire bullet should be removed.

Proposed resolution (September, 2015):

Change 6.2 [basic.def.odr] bullet 6.6 as follows:

- if `D` is a class with an implicitly-declared constructor (15.1 [class.ctor]), it is as if the constructor was implicitly defined in every translation unit where it is odr-used, and the implicit definition in every translation unit shall call the same constructor for a ~~base class or a class member~~ **subobject** of `D`. [*Example:*

```
//translation unit 1:
struct X {
  X(int, int);
  X(int, int, int);
};
X::X(int, int = 0) { }
class D~~: public X~~ {
  X x = 0;
};
D d2;           // X(int, int) called by D()

//translation unit 2:
struct X {
  X(int, int);
  X(int, int, int);
};
X::X(int, int = 0, int = 0) { }
class D~~: public X~~ {
  X x = 0;
};
                // X(int, int, int) called by D();
                // D()'s implicit definition
                // violates the ODR
```

—*end example*]

2104. Internal-linkage `constexpr` references and ODR requirements

Section: 6.2 [basic.def.odr] **Status:** CD4 **Submitter:** James Widman **Date:** 2015-03-17

[Adopted at the February, 2016 meeting.]

In an example like:

```
extern int i;
namespace {
  constexpr int& r = i;
}
inline int f() { return r; }
```

use of `f()` in multiple translation units results in an ODR violation because of use of the internal-linkage reference `r`. It would be helpful if 6.2 [basic.def.odr] paragraph 6 could be amended to "look through" a `constexpr` reference in determining whether an inline function violates the ODR or not.

Proposed resolution (January, 2016):

Change 6.2 [basic.def.odr] bullet 6.2 as follows, dividing the running text into a bulleted list:

...Given such an entity named `D` defined in more than one translation unit, then

- each definition of `D` shall consist of the same sequence of tokens; and
- in each definition of `D`, corresponding names, looked up according to 6.4 [basic.lookup], shall refer to an entity defined within the definition of `D`, or shall refer to the same entity, after overload resolution (16.3 [over.match]) and after matching of partial template specialization (17.9.3 [temp.over]), except that a name can refer to
  - a non-volatile const object with internal or no linkage if the object
    - has the same literal type in all definitions of `D`, ~~and~~
    - ~~the object~~ is initialized with a constant expression (8.20 [expr.const]), ~~and~~
    - ~~the object~~ is not odr-used, and
    - ~~the object~~ has the same value in all definitions of `D`**,**

    or
  - **a reference with internal or no linkage initialized with a constant expression such that the reference refers to the same entity in all definitions of** `D`;

  and
- in each definition of `D`, corresponding entities...

2063. Type/nontype hiding in class scope

Section: 6.3.1 [basic.scope.declarative] **Status:** CD4 **Submitter:** Hubert Tong **Date:** 2014-12-20

[Adopted at the February, 2016 meeting.]

The type/nontype hiding rules ("`struct stat` hack") do not apply in class scope. This is a C compatibility issue:

```
struct A {
  struct B { int x; } b;
  int B;    // Permitted in C
};
```

Since the type/nontype hiding rules exist for C compatibility, should this example be supported?

Proposed resolution (September, 2015):

Change 6.3.1 [basic.scope.declarative] paragraph 4 as follows:

> 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 and function templates; or
> - exactly one declaration shall declare a class name or enumeration name that is not a typedef name and the other declarations shall all refer to the same variable**, non-static data member,** or enumerator, or all refer to functions and function templates; in this case the class name or enumeration name is hidden (6.3.10 [basic.scope.hiding]). [*Note:* A namespace name or a class template name must be unique in its declarative region (10.3.2 [namespace.alias], Clause 17 [temp]). —*end note*]

1875. Reordering declarations in class scope

Section: 6.3.7 [basic.scope.class] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-02-19

[Moved to DR at the May, 2015 meeting.]

The rules for class scope in 6.3.7 [basic.scope.class] paragraph 1 include the following:

2. A name `N` used in a class `S` shall refer to the same declaration in its context and when re-evaluated 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 is ill-formed, no diagnostic is required.

The need for rule #3 is not clear; it would seem that any otherwise-valid reordering would have to violate rule #2 in order to yield a different interpretation. Taken literally, rule #3 would also apply to simply reordering nonstatic data members with no name dependencies at all. Can it be simply removed?

Proposed resolution (June, 2014):

Delete the third item of 6.3.7 [basic.scope.class] paragraph 1 and renumber the succeeding items:

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

1753. *decltype-specifier* in *nested-name-specifier* of destructor

Section: 6.4.3 [basic.lookup.qual] **Status:** CD4 **Submitter:** John Spicer **Date:** 2013-09-18

[Moved to DR at the November, 2014 meeting.]

One of the forms of *pseudo-destructor-name* is

> *nested-name-specifier*$_{opt}$ `~` *type-name*

Presumably the intent of this form is to allow the *nested-name-specifier* to designate a namespace; otherwise the

> *nested-name-specifier*$_{opt}$ *type-name* `::` `~` *type-name*

production would be used.

Since one of the forms of *nested-name-specifier* is

> *decltype-specifier* `::`

one can write something like `p->decltype(x)::~Y()`. However, the lookup rules in 6.4.3 [basic.lookup.qual] paragraph 6 are inappropriate for the *decltype-specifier* case:

> If a *pseudo-destructor-name* (8.2.4 [expr.pseudo]) contains a *nested-name-specifier*, the *type-name*s are looked up as types in the scope designated by the *nested-name-specifier*.

Since this form appears to be useless (use of a *decltype-specifier* is permitted after a `~`, but only with no *nested-name-specifer* — but see issue 1586), perhaps it should be made ill-formed.

Proposed resolution (February, 2014):

Change the grammar in 8.2 [expr.post] paragraph 1 as follows:

pseudo-destructor-name:
nested-name-specifier$_{opt}$ *type-name* `::` `~` *type-name*
nested-name-specifier `template` *simple-template-id* `::` `~` *type-name*
~~*nested-name-specifier*$_{opt}$~~ `~` *type-name*
`~` *decltype-specifier*

1603. Errors resulting from giving unnamed namespaces internal linkage

Section: 6.5 [basic.link] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2013-01-09

[Moved to DR at the November, 2014 meeting.]

In C++03, all namespace-scope names had external linkage unless explicitly declared otherwise (via `static`, `const`, or as a member of an anonymous union). C++11 now specifies that members of an unnamed namespace have internal linkage (see issue 1113). This change invalidated a number of assumptions scattered throughout the Standard that need to be adjusted:

1. 6.5 [basic.link] paragraph 5 says,

   > a member function, static data member, a named class or enumeration of class scope, or an unnamed class or enumeration defined in a class-scope typedef declaration such that the class or enumeration has the typedef name for linkage purposes (10.1.3 [dcl.typedef]), has external linkage if the name of the class has external linkage.

   There is no specification for the linkage of such members of a class with internal linkage. Formally, at least, that leads to the statement in paragraph 8 that such members have no linkage. This omission also contradicts the note in 12.2.1 [class.mfct] paragraph 3:

   > [*Note:* Member functions of a class in namespace scope have external linkage. Member functions of a local class (12.4 [class.local]) have no linkage. See 6.5 [basic.link]. —*end note*]

   as well as the statement in 12.2.3.2 [class.static.data] paragraph 5,

   > `Static` data members of a class in namespace scope have external linkage (6.5 [basic.link]).

2. The footnote in 6.5 [basic.link] paragraph 8 says,

   > A class template always has external linkage, and the requirements of 17.3.1 [temp.arg.type] and 17.3.2 [temp.arg.nontype] ensure that the template arguments will also have appropriate linkage.

   This is incorrect, since templates in unnamed namespaces now have internal linkage and template arguments are no longer required to have external linkage.

3. The statement in 10.1.1 [dcl.stc] paragraph 7 is now false:

   > 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`.

4. The entire treatment of *unique* in 10.3.1.1 [namespace.unnamed] is no longer necessary, and the footnote is incorrect:

   > Although entities in an unnamed namespace might have external linkage, they are effectively qualified by a name unique to their translation unit and therefore can never be seen from any other translation unit.

   Names in unnamed namespaces never have external linkage.

5. According to 14.3 [class.friend] paragraph 4,

   > A function first declared in a friend declaration has external linkage (6.5 [basic.link]).

   This presumably is incorrect for a class that is a member of an unnamed namespace.

6. According to 17 [temp] paragraph 4,

   > A non-member function template can have internal linkage; any other template name shall have external linkage.

   Taken literally, this would mean that a template could not be a member of an unnamed namespace.

Proposed resolution (April, 2013):

1. Change 6.5 [basic.link] paragraph 5 as follows:

In addition, a member function, static data member, a named class or enumeration of class scope, or an unnamed class or enumeration defined in a class-scope typedef declaration such that the class or enumeration has the typedef name for linkage purposes (10.1.3 [dcl.typedef]), has ~~external linkage if the name of the class has external linkage~~ **the same linkage, if any, as the name of the class of which it is a member**.

2. Change the footnote in 6.5 [basic.link] paragraph 8 as follows:

33) A class template ~~always has external linkage, and the requirements of 17.3.1 [temp.arg.type] and 17.3.2 [temp.arg.nontype] ensure that the template arguments will also have appropriate linkage~~ **has the linkage of the innermost enclosing class or namespace in which it is declared**.

3. Change 10.3.1.1 [namespace.unnamed] paragraph 1 as follows:

An *unnamed-namespace-definition* behaves as if it were replaced by

```
inline_opt namespace unique { /* empty body */ }
using namespace unique ;
namespace unique { namespace-body }
```

where `inline` appears if and only if it appears in the *unnamed-namespace-definition*~~,~~ **and** all occurrences of *unique* in a translation unit are replaced by the same identifier, and this identifier differs from all other identifiers in the ~~entire program. [*Footnote:* Although entities in an unnamed namespace might have external linkage, they are effectively qualified by a name unique to their translation unit and therefore can never be seen from any other translation unit. —*end footnote*~~] **translation unit.** [*Example:*...

4. Change the note in 12.2.1 [class.mfct] paragraph 3 as follows:

[*Note:* Member functions of a class in namespace scope have ~~external linkage~~ **the linkage of that class**. Member functions of a local class (12.4 [class.local]) have no linkage. See 6.5 [basic.link]. —*end note*]

5. Change 12.2.3.2 [class.static.data] paragraph 5 as follows:

`Static` data members of a class in namespace scope have ~~external linkage~~ **the linkage of that class** (6.5 [basic.link]).

6. Change 14.3 [class.friend] paragraph 4 as follows:

A function first declared in a friend declaration has ~~external linkage~~ **the linkage of the namespace of which it is a member** (6.5 [basic.link]). Otherwise, the function retains its previous linkage (10.1.1 [dcl.stc]).

7. Change 17 [temp] paragraph 4 as follows:

A template name has linkage (6.5 [basic.link]). ~~A non-member function template can have internal linkage; any other template name shall have external linkage.~~ Specializations (explicit or implicit) of a template that has internal linkage are distinct from all specializations in other translation units...

1686. Which variables are "explicitly declared `const`?"

Section: 6.5 [basic.link] **Status:** CD4 **Submitter:** Daniel Krügler **Date:** 2013-05-17

[Moved to DR at the November, 2014 meeting.]

According to 6.5 [basic.link] paragraph 3,

> A name having namespace scope (6.3.6 [basic.scope.namespace]) has internal linkage if it is the name of
>
> - ...
> - a non-volatile variable that is explicitly declared `const` or `constexpr` and neither explicitly declared `extern` nor previously declared to have external linkage; or
> - ...

It would be more precise and less confusing if the phrase "explicitly declared `const`" were replaced by saying that its type is const-qualified. This change would also allow removal of the reference to `constexpr`, which was added by issue 1112 because `constexpr` variables are implicitly const-qualified but not covered by the "explicitly declared" phrasing.

Proposed resolution (September, 2013):

Change the second bullet of 6.5 [basic.link] paragraph 3 as follows:

- a ~~non-volatile~~ variable ~~that is explicitly declared `const` or `constexpr` and~~ **of non-volatile const-qualified type that is** neither explicitly declared `extern` nor previously declared to have external linkage; or

---

2026. Zero-initialization and `constexpr`

Section: 6.6 [basic.start] **Status:** CD4 **Submitter:** Jason Merrill **Date:** 2014-10-20

[Moved to DR at the October, 2015 meeting.]

According to 6.6.2 [basic.start.static] paragraph 2,

> Variables with static storage duration (6.7.1 [basic.stc.static]) or thread storage duration (6.7.2 [basic.stc.thread]) shall be zero-initialized (11.6 [dcl.init]) before any other initialization takes place.

Does this apply to constant initialization as well? For example, should the following be well-formed, relying on the presumed zero-initialization preceding the constant initialization?

```
constexpr int i = i;
struct s {
  constexpr s() : v(v) { }
  int v;
};
constexpr s s1;
```

Notes from the November, 2014 meeting:

CWG agreed that constant initialization should be considered as happening instead of zero initialization in these cases, making the declarations ill-formed.

Proposed resolution (May, 2015):

1. Rename 6.6.2 [basic.start.static] and make the indicated changes, moving parts of its content to a new section immediately following, as indicated below:

> **3.6.2 ŧStatic initialization ~~of non-local variables~~ [basic.start.~~init~~.static]**
>
> ~~There are two broad classes of named non-local variables: those with static storage duration (6.7.1 [basic.stc.static]) and those with thread storage duration (6.7.2 [basic.stc.thread]). Non-local variables~~ **Variables** with static storage duration are initialized as a consequence of program initiation. ~~Non-local variables~~ **Variables** with thread storage duration are initialized as a consequence of thread execution. Within each of these phases of initiation, initialization occurs as follows.
>
> ~~Variables with static storage duration (6.7.1 [basic.stc.static]) or thread storage duration (6.7.2 [basic.stc.thread]) shall be zero-initialized (11.6 [dcl.init]) before any other initialization takes place.~~ A *constant initializer* for an object `o` is an expression that is a constant expression, except that it may also invoke `constexpr` constructors for o and its subobjects even if those objects are of non-literal class types [*Note:* such a class may have a non-trivial destructor —*end note*]. *Constant initialization* is performed:
>
> - if each full-expression (including implicit conversions) that appears in the initializer of a reference with static or thread storage duration is a constant expression (8.20 [expr.const]) and the reference is bound to a glvalue designating an object with static storage duration, to a temporary object (see 15.2 [class.temporary]) or subobject thereof, or to a function;
> - if an object with static or thread storage duration is initialized by a constructor call, and if the initialization full-expression is a constant initializer for the object;
> - if an object with static or thread storage duration is not initialized by a constructor call and if either the object is value-initialized or every full-expression that appears in its initializer is a constant expression.
>
> **If constant initialization is not performed, a variable with static storage duration (6.7.1 [basic.stc.static]) or thread storage duration (6.7.2 [basic.stc.thread]) is zero-initialized (11.6 [dcl.init]).** Together, zero-initialization and constant initialization are called *static initialization*; all other initialization is *dynamic initialization*. Static initialization shall be performed before any dynamic initialization takes place. ~~Dynamic initialization of a non-local variable with static storage duration is *unordered* if the variable is an implicitly or explicitly instantiated specialization, and otherwise is *ordered* [*Note:* an explicitly specialized static data member or variable template specialization has ordered initialization. —*end note*]. Variables with ordered initialization defined within a single translation unit shall be initialized in the order of their definitions in the translation unit. If a program starts a thread (*33.3* [thread.threads]), the subsequent initialization of a variable is unsequenced with respect to the initialization of a variable defined in a different translation unit. Otherwise, the initialization of a variable is indeterminately sequenced with respect to the initialization of a variable defined in a different translation unit. If a program starts a thread, the subsequent unordered initialization of a variable is unsequenced with respect to every other dynamic initialization. Otherwise, the unordered initialization of a variable is indeterminately sequenced with respect to every other dynamic initialization. [*Note:* This definition permits initialization of a sequence of ordered variables concurrently with another sequence. —*end note*]~~ [*Note:* The **dynamic** initialization of **non-local variables is described in 3.6.3 [basic.start.dynamic]; that of** local static variables is described in 9.7 [stmt.dcl]. —*end note*]
>
> An implementation is permitted to perform the initialization of a ~~non-local~~ variable with static **or thread** storage duration as a static initialization even if such initialization is not required to be done statically, provided that
>
> - the dynamic version of the initialization does not change the value of any other object of ~~namespace scope~~ **static or thread storage duration** prior to its initialization, and
> - the static version of the initialization produces the same value in the initialized variable as would be produced by the dynamic initialization if all variables not required to be initialized statically were initialized dynamically.

[*Note:* As a consequence, if the initialization of an object `obj1` refers to an object `obj2` of namespace scope potentially requiring dynamic initialization and defined later in the same translation unit, it is unspecified whether the value of `obj2` used will be the value of the fully initialized `obj2` (because `obj2` was statically initialized) or will be the value of `obj2` merely zero-initialized. For example,

```
inline double fd() { return 1.0; }
extern double d1;
double d2 = d1;   // unspecified:
                  // may be statically initialized to 0.0 or
                  // dynamically initialized to 0.0 if d1 is
                  // dynamically initialized, or 1.0 otherwise
double d1 = fd(); // may be initialized statically or dynamically to 1.0
```

—*end note*]

2. Insert a new section after 6.6.2 [basic.start.static]:

3.6.3 Dynamic initialization of non-local variables [basic.start.dynamic]

3. Move part of 6.6.2 [basic.start.static] paragraph 2 as paragraph 1 of the new section:

Dynamic initialization of a non-local variable with static storage duration is unordered if the variable is an implicitly or explicitly instantiated specialization, and otherwise is ordered [*Note:* an explicitly specialized static data member or variable template specialization has ordered initialization. —*end note*]. Variables with ordered initialization defined within a single translation unit shall be initialized in the order of their definitions in the translation unit. If a program starts a thread (33.3 [thread.threads]), the subsequent initialization of a variable is unsequenced with respect to the initialization of a variable defined in a different translation unit. Otherwise, the initialization of a variable is indeterminately sequenced with respect to the initialization of a variable defined in a different translation unit. If a program starts a thread, the subsequent unordered initialization of a variable is unsequenced with respect to every other dynamic initialization. Otherwise, the unordered initialization of a variable is indeterminately sequenced with respect to every other dynamic initialization. [*Note:* This definition permits initialization of a sequence of ordered variables concurrently with another sequence. —*end note*]

4. Move paragraphs 4-6 of 6.6.2 [basic.start.static] as paragraphs 2-4 of the new section:

It is implementation-defined whether the dynamic initialization of a non-local variable with static storage duration is done before the first statement of `main`. If the initialization is deferred to some point in time after the first statement of `main`, it shall occur before the first odr-use (6.2 [basic.def.odr]) of any function or variable defined in the same translation unit as the variable to be initialized. [*Footnote:* A non-local variable with static storage duration having initialization with side-effects must be initialized even if it is not odr-used (6.2 [basic.def.odr], 6.7.1 [basic.stc.static]). —*end footnote*] [*Example:*

```
// - File 1 -
#include "a.h"
#include "b.h"
B b;
A::A(){
  b.Use();
}

// - File 2 -
#include "a.h"
A a;

// - File 3 -
#include "a.h"
#include "b.h"
extern A a;
extern B b;
int main() {
  a.Use();
  b.Use();
}
```

It is implementation-defined whether either `a` or `b` is initialized before `main` is entered or whether the initializations are delayed until `a` is first odr-used in `main`. In particular, if `a` is initialized before `main` is entered, it is not guaranteed that `b` will be initialized before it is odr-used by the initialization of `a`, that is, before `A::A` is called. If, however, `a` is initialized at some point after the first statement of `main`, `b` will be initialized prior to its use in `A::A`. —*end example*]

It is implementation-defined whether the dynamic initialization of a non-local variable with static or thread storage duration is done before the first statement of the initial function of the thread. If the initialization is deferred to some point in time after the first statement of the initial function of the thread, it shall occur before the first odr-use (6.2 [basic.def.odr]) of any variable with thread storage duration defined in the same translation unit as the variable to be initialized.

If the initialization of a non-local variable with static or thread storage duration exits via an exception, `std::terminate` is called (18.5.1 [except.terminate]).

5. Change 9.7 [stmt.dcl] paragraph 4 as follows:

~~The zero-initialization (11.6 [dcl.init]) of all block-scope variables with static storage duration (6.7.1 [basic.stc.static]) or thread storage duration (6.7.2 [basic.stc.thread]) is performed before any other initialization takes place. Constant initialization (6.6.2 [basic.start.static]) of a block-scope entity with static storage duration, if applicable, is performed before its block is first entered. An implementation is permitted to perform early initialization of other block-scope~~

~~variables with static or thread storage duration under the same conditions that an implementation is permitted to statically initialize a variable with static or thread storage duration in namespace scope (6.6.2 [basic.start.static]). Otherwise such a variable is initialized~~ **Dynamic initialization of a block-scope variable with static storage duration (6.7.1 [basic.stc.static]) or thread storage duration (6.7.2 [basic.stc.thread]) is performed** the first time control passes...

Editing note: all existing cross-references to 6.6.2 [basic.start.static] must be examined to determine which of the two current sections should be targeted.

1886. Language linkage for `main()`

Section: 6.6.1 [basic.start.main] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-03-04

[Moved to DR at the May, 2015 meeting.]

There does not appear to be any restriction on giving `main()` an explicit language linkage, but it should probably be either ill-formed or conditionally-supported.

Proposed resolution (November, 2014):

1. Change 6.6.1 [basic.start.main] paragraph 2 as follows:

   An implementation shall not predefine the `main` function. This function shall not be overloaded. ~~It~~ **Its type shall have C++ language linkage and it** shall have a declared return type of type `int`, but otherwise its type is implementation-defined. An implementation shall allow both...

2. Change 6.6.1 [basic.start.main] paragraph 3 as follows:

   The function `main` shall not be used within a program. The linkage (6.5 [basic.link]) of `main` is implementation-defined. A program that defines `main` as deleted or that declares `main` to be `inline`, `static`, or `constexpr` is ill-formed. **The `main` function shall not be declared with a *linkage-specification* (10.5 [dcl.link]). A program that declares a variable `main` at global scope or that declares the name `main` with C language linkage (in any namespace) is ill-formed.** The name `main` is not otherwise reserved...

1744. Unordered initialization for variable template specializations

Section: 6.6.2 [basic.start.static] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2013-09-03

[Moved to DR at the November, 2014 meeting.]

According to 6.6.2 [basic.start.static] paragraph 2,

> Definitions of explicitly specialized class template static data members have ordered initialization. Other class template static data members (i.e., implicitly or explicitly instantiated specializations) have unordered initialization.

This is not clear whether it is referring to static data members of explicit specializations of class templates or to explicit specializations of static data members of class template specializations. It also does not apply to static data member templates and non-member variable templates.

Proposed resolution (February, 2014):

Change 6.6.2 [basic.start.static] paragraph 2 as follows:

...Dynamic initialization of a non-local variable with static storage duration is ~~either ordered or unordered. Definitions of explicitly specialized class template static data members have ordered initializa-tion. Other class template static data members (i.e., implicitly or explicitly instantiated specializations) have unordered initialization. Other non-local variables with static storage duration have ordered initialization~~ ***unordered*** **if the variable is an implicitly or explicitly instantiated specialization, and otherwise is *ordered* [*Note:* an explicitly specialized static data member or variable template specialization has ordered initialization. —*end note*]**. Variables with ordered initialization...

1834. Constant initialization binding a reference to an xvalue

Section: 6.6.2 [basic.start.static] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-01-13

[Moved to DR at the November, 2014 meeting.]

According to 6.6.2 [basic.start.static] paragraph 2,

> *Constant initialization* is performed:

- if each full-expression (including implicit conversions) that appears in the initializer of a reference with static or thread storage duration is a constant expression (8.20 [expr.const]) and the reference is bound to an lvalue designating an object with static storage duration or to a temporary (see 15.2 [class.temporary]);
- ...

This wording should also permit the reference to be bound to an xvalue, e.g., a subobject of a temporary, and not just to a complete temporary.

Proposed resolution (February, 2014):

Change 6.6.2 [basic.start.static] paragraph 2 as follows (note that this resolution incorporates the overlapping change from the resolution of issue 1299)::

...*Constant initialization* is performed:

- if each full-expression (including implicit conversions) that appears in the initializer of a reference with static or thread storage duration is a constant expression (8.20 [expr.const]) and the reference is bound to ~~an lvalue~~ **a glvalue** designating an object with static storage duration, to a temporary **object** (see 15.2 [class.temporary]) **or subobject thereof**, or to a function;
- ...

2012. Lifetime of references

Section: 6.7 [basic.stc] **Status:** CD4 **Submitter:** Mike Miller **Date:** 2014-09-29

[Adopted at the February, 2016 meeting.]

According to 6.7 [basic.stc] paragraph 3,

> The storage duration categories apply to references as well. The lifetime of a reference is its storage duration.

This is clearly not correct; references can have static storage duration but be dynamically initialized. Consider an example like:

```
extern int& r1;
int& f();
int& r2 = r1;  // #1
int& r1 = f();
int i = r2;    // #2
```

`r1` is not initialized until after its use at #1, so the initialization of `r2` should produce undefined behavior, as should the use of `r2` at #2.

The description of the lifetime of a reference should be deleted from 6.7 [basic.stc] and it should be described properly in 6.8 [basic.life].

Proposed resolution (September, 2015):

1. Change 6.7 [basic.stc] paragraph 3 as follows:

   The storage duration categories apply to references as well. ~~The lifetime of a reference is its storage duration.~~

2. Change 6.8 [basic.life] paragraph 1 as follows:

   The *lifetime* of an object **or reference** is a runtime property of the object **or reference**. An object is said to have...

3. Add the following as a new paragraph following 6.7 [basic.stc] paragraph 2:

   [*Note:* The lifetime of an array object starts as soon as storage with proper size and alignment is obtained, and its lifetime ends when the storage which the array occupies is reused or released. 15.6.2 [class.base.init] describes the lifetime of base and member subobjects. — *end note*]

   The lifetime of a reference begins when its initialization is complete. The lifetime of a reference ends as if it were a scalar object.

4. Change 6.8 [basic.life] paragraph 3 as follows:

   The properties ascribed to objects **and references** throughout this International Standard apply for a given object **or reference** only during its lifetime. [*Note:*...

Change 8 [expr] paragraph 5 as follows:

> If an expression initially has the type "reference to `T`" (11.3.2 [dcl.ref], 11.6.3 [dcl.init.ref]), the type is adjusted to `T` prior to any further analysis. The expression designates the object or function denoted by the reference, and the expression is an lvalue or an xvalue, depending on the expression. **[*Note:* Before the lifetime of the reference has started or after it has ended, the behavior is undefined (see 6.8 [basic.life]). — *end note*]**

Drafting note: there is no change to 6.8 [basic.life] paragraph 4:

A program may end the lifetime of any object by reusing the storage which the object occupies or by explicitly calling the destructor for an object of a class type with a non-trivial destructor. For an object of a class type...

1956. Reuse of storage of automatic variables

Section: 6.7.3 [basic.stc.auto] **Status:** CD4 **Submitter:** Daniel Krügler **Date:** 2014-06-29

[Moved to DR at the May, 2015 meeting.]

According to 6.7.3 [basic.stc.auto] paragraph 3,

> If a variable with automatic storage duration has initialization or a destructor with side effects, it shall not be destroyed before the end of its block, nor shall it be eliminated as an optimization even if it appears to be unused, except that a class object or its copy/move may be eliminated as specified in 15.8 [class.copy].

This is intended to be a requirement for the implementation, but it could be read as prohibiting the reuse of the storage of an automatic variable by the program using a placement *new-expression*.

Proposed resolution (November, 2014):

Change 6.7.3 [basic.stc.auto] paragraph 3 as follows:

> If a variable with automatic storage duration has initialization or a destructor with side effects, ~~it~~ **an implementation** shall not ~~be destroyed~~ **destroy it** before the end of its block~~,~~ nor ~~shall it be eliminated~~ **eliminate it** as an optimization**,** even if it appears to be unused, except that a class object or its copy/move may be eliminated as specified in 15.8 [class.copy].

1338. Aliasing and allocation functions

Section: 6.7.4.1 [basic.stc.dynamic.allocation] **Status:** CD4 **Submitter:** Jason Merrill **Date:** 2011-08-03

[Moved to DR at the November, 2014 meeting.]

In 6.7.4.1 [basic.stc.dynamic.allocation] paragraph 2, allocation functions are constrained to return a pointer that is different from any previously returned pointer that has not been passed to a deallocation function. This does not, for instance, prohibit returning a pointer to storage that is part of another object, for example, a pool of storage. The potential implications of this for aliasing should be spelled out.

(See also issues 1027 and 1116.)

Additional note (March, 2013):

One possibility to allow reasonable optimizations would be to require that allocation packages hide their storage in file-static variables, perhaps by adding wording such as:

> Furthermore, `p0` shall point to an object distinct from any other object that is accessible outside the implementation of the allocation and deallocation functions.

Additional note, April, 2013:

Concern was expressed that a pool class might provide an interface for iterating over all the pointers that were given out from the pool, and this usage should be supported.

Notes from the September, 2013 meeting:

CWG agreed that changes for this issue should apply only to non-placement forms.

Proposed resolution (February, 2014):

Change 6.7.4.1 [basic.stc.dynamic.allocation] paragraph 2 as follows:

> ...If the request succeeds, the value returned shall be a non-null pointer value (7.11 [conv.ptr]) `p0` different from any previously returned value `p1`, unless that value `p1` was subsequently passed to an `operator delete`. **Furthermore, for the library allocation functions in 21.6.2.1 [new.delete.single] and 21.6.2.2 [new.delete.array], `p0` shall point to a block of storage disjoint from the storage for any other object accessible to the caller.** The effect of indirecting through a pointer returned as a request for zero size is undefined.[36]

2019. Member references omitted from description of storage duration

Section: 6.7.5 [basic.stc.inherit] **Status:** CD4 **Submitter:** David Krauss **Date:** 2014-10-08

[Moved to DR at the October, 2015 meeting.]

According to 6.7.5 [basic.stc.inherit] paragraph 1,

> The storage duration of member subobjects, base class subobjects and array elements is that of their complete object (4.5 [intro.object]).

This wording does not cover member references, which should also have the same storage duration as the object of which they are a member.

Proposed resolution (May, 2015):

Change 6.7.5 [basic.stc.inherit] paragraph 1 as follows:

> The storage duration of ~~member~~ subobjects~~, base class subobjects~~ and ~~array elements~~ **reference members** is that of their complete object (4.5 [intro.object]).

1116. Aliasing of union members

Section: 6.8 [basic.life] **Status:** CD4 **Submitter:** US **Date:** 2010-08-02

[Adopted at the June, 2016 meeting as document P0137R1.]

N3092 comment US 27

Related to issue 1027, consider:

```
int f() {
  union U { double d; } u1, u2;
  (int&)u1.d = 1;
  u2 = u1;
  return (int&)u2.d;
}
```

Does this involve undefined behavior? 6.8 [basic.life] paragraph 4 seems to say that it's OK to clobber `u1` with an `int` object. Then union assignment copies the object representation, possibly creating an `int` object in `u2` and making the return statement well-defined. If this is well-defined, compilers are significantly limited in the assumptions they can make about type aliasing. On the other hand, the variant where `U` has an array of `unsigned char` member must be well-defined in order to support `std::aligned_storage`.

Suggested resolution: Clarify that this case is undefined, but that adding an array of `unsigned char` to union `U` would make it well-defined — if a storage location is allocated with a particular type, it should be undefined to create an object in that storage if it would be undefined to access the stored value of the object through the allocated type.

(See also issues 1027 and 1338.)

Proposed resolution (August, 2010):

1. Change 6.8 [basic.life] paragraph 1 as follows:

   > ...The lifetime of an object of type `T` begins when **storage with the proper alignment and size for type `T` is obtained, and either**:
   >
   > - ~~storage with the proper alignment and size for type `T` is obtained, and~~
   > - if the object has non-trivial initialization, its initialization is complete~~.~~**, or**
   > - **if `T` is trivially copyable, the object representation of another `T` object is copied into the storage.**
   >
   > The lifetime of an object of type `T` ends...

2. Change 6.8 [basic.life] paragraph 4 as follows:

   > A program may end the lifetime of any object by reusing the storage which the object occupies or by explicitly calling the destructor for an object of a class type with a non-trivial destructor. For an object of a class type with a non-trivial destructor, the program is not required to call the destructor explicitly before the storage which the object occupies is reused or released; however, if there is no explicit call to the destructor or if a *delete-expression* (8.3.5 [expr.delete]) is not used to release the storage, the destructor shall not be implicitly called and any program that depends on the side effects produced by the destructor has undefined behavior. **If a program obtains storage for an object of a particular type `A` (e.g. with a variable definition or *new-expression*) and later reuses that storage for an object of another type `B` such that accessing the stored value of the `B` object through a glvalue of type `A` would have undefined behavior (6.10 [basic.lval]), the behavior is undefined. [*Example:***
   >
   > ```
   > int i;
   > (double&)i = 1.0; // undefined behavior
   >
   > struct S { unsigned char alignas(double) ar[sizeof (double)]; } s;
   > (double&)s = 1.0; // OK, can access stored double through s because it has an unsigned char subobject
   > ```

—end example]

3. Change 6.10 [basic.lval] paragraph 10 as follows:

> If a program attempts to access the stored value of an object through a glvalue of other than one of the following types the behavior is undefined[52]**:**
>
> - the dynamic type of the object,
> - a cv-qualified version of the dynamic type of the object,
> - a type similar (as defined in 7.5 [conv.qual]) to the dynamic type of the object,
> - a type that is the signed or unsigned type corresponding to the dynamic type of the object,
> - a type that is the signed or unsigned type corresponding to a cv-qualified version of the dynamic type of the object,
> - **a** `char` **or** `unsigned char` **type,**
> - an aggregate or union type that includes one of the aforementioned types among its elements**, bases,** or non-static data members (including, recursively, an element**, base,** or non-static data member of a subaggregate**, base,** or contained union)**;.**
> - ~~a type that is a (possibly cv-qualified) base class type of the dynamic type of the object,~~
> - ~~a `char` or `unsigned char` type.~~

This resolution also resolves issue 1027.

Additional note (August, 2012):

Concerns have been raised regarding the interaction of this change with facilities like `std::aligned_storage` and memory pools. Care must be taken to achieve the proper balance between supporting type-based optimization techniques and allowing practical storage management.

Additional note (January, 2013):

Several questions have been raised about the wording above . In particular:

1. Since aggregates and unions cannot have base classes, why are base classes mentioned?
2. Since unions can now have special member functions, is it still valid to assume that they alias all their member types?
3. Shouldn't standard-layout classes also be considered and not just aggregates?

Additional note, February, 2014:

According to 4.5 [intro.object] paragraph 1, an object (i.e., a "region of storage") is created by one of only three means:

> An object is created by a definition (6.1 [basic.def]), by a *new-expression* (8.3.4 [expr.new]) or by the implementation (15.2 [class.temporary]) when needed. The properties of an object are determined when the object is created.

This does not allow for obtaining the storage in other ways, such as via `malloc`, in determining the lifetime of an object with vacuous initialization (6.8 [basic.life] paragraph 1).

In addition, 6.8 [basic.life] paragraph 1 does not require the storage obtained for an object of type `T` to be accessed via an lvalue of type `T` in order to be considered an object of that type. The treatment of "effective type" by C may be helpful here.

Additional note, May, 2015:

We never say what the active member of a union is, how it can be changed, and so on. The Standard doesn't make clear whether the following is valid:

```
union U { int a; short b; } u = { 0 };
int x = u.a; // presumably this is OK, but we never say that a is the active member
u.b = 0;     // not clear whether this is valid
```

The closest we come to talking about this is the non-normative example in 12.3 [class.union] paragraph 4, which suggests that a placement `new` is needed.

It's also not clear whether `a` has two subobjects or only one (corresponding to the active member).

1284. Should the lifetime of an array be independent of that of its elements?

Section: 6.8 [basic.life] **Status:** CD4 **Submitter:** Gabriel Dos Reis **Date:** 2011-04-02

[Adopted at the February, 2016 meeting.]

The note in 6.8 [basic.life] paragraph 2 reads,

[*Note:* The lifetime of an array object starts as soon as storage with proper size and alignment is obtained, and its lifetime ends when the storage which the array occupies is reused or released. 15.6.2 [class.base.init] describes the lifetime of base and member subobjects. —*end note*]

This wording reflects an earlier version of paragraph 1 that deferred the start of an object's lifetime only for initialization of objects of class type. The note simply emphasized the implication that that the lifetime of a POD type or an array began immediately, even if lifetime of an array's elements began later.

The decomposition of POD types removed the mention of PODs, leaving only the array types, and when the normative text was changed to include aggregates whose members have non-trivial initialization, the note was overlooked.

It is not clear whether it would be better to update the note to emphasize the distinction between aggregates with non-trivial initialization and those without or to delete it entirely.

A possible related normative change to consider is whether the specification of paragraph 1 is sufficiently clear with respect to multidimensional arrays. The current definition of "non-trivial initialization" is:

> An object is said to have non-trivial initialization if it is of a class or aggregate type and it or one of its members is initialized by a constructor other than a trivial default constructor.

Presumably the top-level array of an N-dimensional array whose ultimate element type is a class type with non-trivial initialization would also have non-trivial initialization, but it's not clear that this wording says that.

A more radical change that came up in the discussion was whether the undefined behavior resulting from an lvalue-to-rvalue conversion of an uninitialized object in 7.1 [conv.lval] paragraph 1 would be better dealt with as a lifetime violation instead.

Proposed resolution (October, 2015):

Change 6.8 [basic.life] paragraphs 1 and 2 as follows:

> The *lifetime* of an object is a runtime property of the object. An object is said to have *non-vacuous* initialization if it is of a class or aggregate type and it or one of its ~~members~~ **subobjects** is initialized by a constructor other than a trivial default constructor. [*Note:* initialization...
>
> [*Note:* ~~The lifetime of an array object starts as soon as storage with proper size and alignment is obtained, and its lifetime ends when the storage which the array occupies is reused or released.~~ 15.6.2 [class.base.init] describes the lifetime of base and member subobjects. —*end note*]

1751. Non-trivial operations vs non-trivial initialization

Section: 6.8 [basic.life] **Status:** CD4 **Submitter:** Nico Josuttis **Date:** 2013-09-15

[Moved to DR at the November, 2014 meeting.]

The description of `is_trivially_constructible` in 23.15.4.3 [meta.unary.prop] paragraph 3 says,

> `is_constructible<T, Args...>::value` is `true` and the variable definition for `is_constructible`, as defined below, is known to call no operation that is not trivial (6.9 [basic.types], 15 [special]).

This risks confusion when compared with the wording in 6.8 [basic.life] paragraph 1,

> An object is said to have non-trivial initialization if it is of a class or aggregate type and it or one of its members is initialized by a constructor other than a trivial default constructor. [*Note:* initialization by a trivial copy/move constructor is non-trivial initialization. —*end note*]

The latter was written long before "trivial" became an important concept in its own right and uses the term differently from how it is used elsewhere in the Standard (as evidenced by the note referring to copy/move construction). The intent is to capture the idea that there is some actual initialization occurring; it should be rephrased to avoid the potential of confusion with the usage of "trivial" elsewhere.

Proposed resolution (February, 2014):

Change 6.8 [basic.life] paragraph 1 as follows:

> The *lifetime* of an object is a runtime property of the object. An object is said to have ~~non-trivial initialization~~ ***non-vacuous initialization*** if it is of a class or aggregate type and it or one of its members is initialized by a constructor other than a trivial default constructor. [*Note:* initialization by a trivial copy/move constructor is ~~non-trivial~~ **non-vacuous** initialization. —*end note*] The lifetime of an object of type `T` begins when:
>
> - storage with the proper alignment and size for type `T` is obtained, and
> - if the object has ~~non-trivial~~ **non-vacuous** initialization, its initialization is complete.
>
> The lifetime of an object...

1776. Replacement of class objects containing reference members

Section: 6.8 [basic.life] **Status:** CD4 **Submitter:** Finland **Date:** 2013-09-24

[Adopted at the June, 2016 meeting as document P0137R1.]

N3690 comment FI 15

The rules given in 6.8 [basic.life] paragraph 7 for when an object's lifetime can be ended and a new object created in its storage include the following restriction:

> the type of the original object is not const-qualified, and, if a class type, does not contain any non-static data member whose type is const-qualified or a reference type

The intent of this restriction is to permit optimizers to rely upon the original values of `const` and reference members in their analysis of subsequent code. However, this makes it difficult to implement certain desirable functionality such as `optional<T>`; see this discussion for more details.

This rule should be reconsidered, at least as far as it applies to unions. If it is decided to keep the rule, acceptable programming techniques for writing safe code when replacing such objects should be outlined in a note.

(See also issue 1404, which will become moot if the restriction is lifted.)

Notes from the October, 2015 meeting:

See also paper P0137 and issue 2182.

Notes from the February, 2017 meeting.

The resolution of this issue also resolves issue 636.

1951. Cv-qualification and literal types

Section: 6.9 [basic.types] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-06-19

[Moved to DR at the May, 2015 meeting.]

6.9 [basic.types] paragraph 10 isn't clear whether a const-qualified class type can be a literal type, and the same for `const void`.

Proposed resolution (April, 2015):

Change 6.9 [basic.types] paragraph 10 as follows:

> A type is a *literal type* if it is:
>
> - **possibly cv-qualified** `void`; or
> - a scalar type; or
> - a reference type; or
> - an array of literal type; or
> - a **possibly cv-qualified** class type (Clause 12 [class]) that has all of the following properties:
>
> - ...

2096. Constraints on literal unions

Section: 6.9 [basic.types] **Status:** CD4 **Submitter:** Agustín K-ballo Bergé **Date:** 2015-03-11

[Adopted at the February, 2016 meeting.]

According to 6.9 [basic.types] bullet 10.5.3, all the members of a class type must be of non-volatile literal types. This seems overly constraining for unions; it would seem to be sufficient if at least one of its non-static members were of a literal type.

Proposed resolution (September, 2015):

Change 6.9 [basic.types] bullet 10.5 as follows:

> A type is a *literal type* if it is:

- ...
- a possibly cv-qualified class type (Clause 12 [class]) that has all of the following properties:
  - it has a trivial destructor,
  - it is an aggregate type (11.6.1 [dcl.init.aggr]) or has at least one constexpr constructor or constructor template that is not a copy or move constructor, ~~and~~
  - **if it is a union, at least one of its non-static data members is of non-volatile literal type, and**
  - **if it is not a union,** all of its non-static data members and base classes are of non-volatile literal types.

1797. Are all bit patterns of `unsigned char` distinct numbers?

Section: 6.9.1 [basic.fundamental] **Status:** CD4 **Submitter:** Tony van Eerd **Date:** 2013-10-02

[Moved to DR at the November, 2014 meeting.]

According to 6.9.1 [basic.fundamental] paragraph 1,

> For unsigned narrow character types, all possible bit patterns of the value representation represent numbers.

Presumably the intent is that each distinct bit pattern represents a different number, but this should be made explicit.

Proposed resolution (February, 2014):

Change 6.9.1 [basic.fundamental] paragraph 1 as follows:

> ...For unsigned narrow character types, ~~all~~ **each** possible bit ~~patterns~~ **pattern** of the value representation ~~represent numbers~~ **represents a distinct number**. These requirements...

2006. Cv-qualified `void` types

Section: 6.9.2 [basic.compound] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-09-16

[Moved to DR at the October, 2015 meeting.]

According to 6.9.2 [basic.compound] paragraph 3,

> The type of a pointer to `void` or a pointer to an object type is called an *object pointer type*. [*Note:* A pointer to `void` does not have a pointer-to-object type, however, because `void` is not an object type. —*end note*]

This wording excludes cv-qualified `void` types. There are other references in the Standard to "`void` type" that are apparently intended to include cv-qualified versions as well.

Proposed resolution (May, 2015):

1. Change 6.9 [basic.types] paragraph 5 as follows:

   > ...Incompletely-defined object types and ~~the void types~~ ***cv*** `void` are incomplete types (6.9.1 [basic.fundamental])...

2. Change 6.9 [basic.types] paragraph 8 as follows:

   > An object type is a (possibly cv-qualified) type that is not a function type, not a reference type, and not ~~a void type~~ ***cv*** `void`.

3. Change 6.9.1 [basic.fundamental] paragraph 9 as follows:

   > ~~The `void` type has an empty set of values. The~~ **A type *cv*** `void` ~~type~~ is an incomplete type that cannot be completed**; such a type has an empty set of values**. It is used as the return type for functions that do not return a value. Any expression can be explicitly converted to type *cv* `void` (8.4 [expr.cast]). An expression of type ***cv*** `void` shall be used only as an expression statement (9.2 [stmt.expr]), as an operand of a comma expression (8.19 [expr.comma]), as a second or third operand of `?:` (8.16 [expr.cond]), as the operand of `typeid`, `noexcept`, or `decltype`, as the expression in a return statement (9.6.3 [stmt.return]) for a function with the return type ***cv*** `void`, or as the operand of an explicit conversion to type *cv* `void`.

4. Change bullet 1.3 of 6.9.2 [basic.compound] as follows:

   - *pointers* to ***cv*** `void` or objects or functions (including static members of classes) of a given type, 11.3.1 [dcl.ptr];

5. Change 6.9.2 [basic.compound] paragraph 3 as follows:

   > The type of a pointer to ***cv*** `void` or a pointer to an object type is called an *object pointer type*. [*Note:*...

636. Dynamic type of objects and aliasing

Section: 6.10 [basic.lval] **Status:** CD4 **Submitter:** Gabriel Dos Reis **Date:** 23 May 2007

The aliasing rules given in 6.10 [basic.lval] paragraph 10 rely on the concept of "dynamic type." The problem is that the dynamic type of an object often cannot be determined (or even sufficiently constrained) at the point at which an optimizer needs to be able to determine whether aliasing might occur or not. For example, consider the function

```
void foo(int* p, double* q) {
    *p = 42;
    *q = 3.14;
}
```

An optimizer, on the basis of the existing aliasing rules, might decide that an `int*` and a `double*` cannot refer to the same object and reorder the assignments. This reordering, however, could result in undefined behavior if the function `foo` is called as follows:

```
void goo() {
   union {
      int i;
      double d;
   } t;

   t.i = 12;

   foo(&t.i, &t.d);

   cout << t.d << endl;
};
```

Here, the reference to `t.d` after the call to `foo` will be valid only if the assignments in `foo` are executed in the order in which they were written; otherwise, the union will contain an `int` object rather than a `double`.

One possibility would be to require that if such aliasing occurs, it be done only via member names and not via pointers.

Notes from the July, 2007 meeting:

This is the same issue as C's DR236. The CWG expressed a desire to address the issue the same way C99 does. The issue also occurs in C++ when placement new is used to end the lifetime of one object and start the lifetime of a different object occupying the same storage.

Proposed resolution (March, 2017):

This issue is resolved by the resolution of issue 1776.

2122. Glvalues of `void` type

Section: 6.10 [basic.lval] **Status:** CD4 **Submitter:** CWG **Date:** 2015-05-05

[Adopted at the February, 2016 meeting.]

According to 6.10 [basic.lval] paragraph 4,

> Unless otherwise indicated (8.2.2 [expr.call]), prvalues shall always have complete types or the `void` type; in addition to these types, glvalues can also have incomplete types.

This wording inadvertently implies that glvalues can have type `void`, which is not correct.

Proposed resolution (January, 2016):

Change 6.10 [basic.lval] paragraph 4 as follows:

> Unless otherwise indicated (8.2.2 [expr.call]), ~~prvalues~~ **a prvalue** shall always have complete ~~types~~ **type** or the `void` type~~; in addition to these types, glvalues can also have incomplete types~~. **A glvalue shall not have type *cv* `void`.** [*Note:* ~~class~~ **A glvalue may have complete or incomplete non-`void` type. Class** and array prvalues can have cv-qualified types; other prvalues always have cv-unqualified types. See Clause 8 [expr]. —*end note*]

1981. Implicit contextual conversions and `explicit`

Section: 7 [conv] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-08-08

[Moved to DR at the October, 2015 meeting.]

According to 7 [conv] paragraph 5,

> Certain language constructs require conversion to a value having one of a specified set of types appropriate to the construct. An expression `e` of class type `E` appearing in such a context is said to be *contextually implicitly converted to* a specified type `T` and is well-formed if and only if `e` can be implicitly converted to a type `T` that is determined as follows: `E` is searched for conversion functions whose return type is *cv* `T` or reference to *cv* `T` such that `T` is allowed by the context. There shall be exactly one such `T`.

This description leaves open two questions: first, can `explicit` conversion functions be used for this conversion? Second, assuming that they cannot, is the restriction to "exactly one such `T`" enforced before or after exclusion of `explicit` conversion functions?

Notes from the November, 2014 meeting:

CWG felt that `explicit` conversion functions should be removed from consideration before determining the set of types for the conversion.

Proposed resolution (May, 2015):

Change 7 [conv] paragraph 5 as follows:

> ...An expression `e` of class type `E` appearing in such a context is said to be *contextually implicitly converted* to a specified type `T` and is well-formed if and only if `e` can be implicitly converted to a type `T` that is determined as follows: `E` is searched for **non-explicit** conversion functions whose return type is *cv* `T` or reference to *cv* `T` such that `T` is allowed by the context. There shall be exactly one such `T`.

2140. Lvalue-to-rvalue conversion of `std::nullptr_t`

Section: 7.1 [conv.lval] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2015-06-12

[Adopted at the February, 2016 meeting.]

The current rules in 7.1 [conv.lval] paragraph 2 do not require fetching the value in memory of an object of type `std::nullpt_t` in order to produce its prvalue. This choice has implications that may not have been considered for questions like whether use of a `std::nullptr_t` that is an inactive member of a union results in undefined behavior or whether a volatile `std::nullptr_t` variable in a discarded-value expression produces a side effect.

Proposed resolution (January, 2016):

Change 7.1 [conv.lval] bullet 2.3 as follows:

> ...In all other cases, the result of the conversion is determined according to the following rules:
>
> - If `T` is (possibly cv-qualified) `std::nullptr_t`, the result is a null pointer constant (7.11 [conv.ptr]). **[*Note:* Since no value is fetched from memory, there is no side effect for a volatile access (4.6 [intro.execution]), and an inactive member of a union (12.3 [class.union]) may be accessed. —*end note*]**
> - ...

---

330. Qualification conversions and pointers to arrays of pointers

Section: 7.5 [conv.qual] **Status:** CD4 **Submitter:** Roger Orr **Date:** 2 Jan 2002

[Moved to DR as N4261 at the November, 2014 meeting.]

Section 7.5 [conv.qual] covers the case of multi-level pointers, but does not appear to cover the case of pointers to arrays of pointers. The effect is that arrays are treated differently from simple scalar values.

Consider for example the following code: (from the thread "Pointer to array conversion question" begun in comp.lang.c++.moderated)

```
int main()
{
   double *array2D[2][3];

   double       *       (*array2DPtr1)[3] = array2D;     // Legal
   double       * const (*array2DPtr2)[3] = array2DPtr1; // Legal
   double const * const (*array2DPtr3)[3] = array2DPtr2; // Illegal
}
```

and compare this code with:-

```
int main()
{
   double *array[2];

   double       *       *ppd1 = array; // legal
   double       * const *ppd2 = ppd1;  // legal
   double const * const *ppd3 = ppd2;  // certainly legal (4.4/4)
}
```

The problem appears to be that the pointed to types in example 1 are unrelated since nothing in the relevant section of the standard covers it - 7.5 [conv.qual] does not mention conversions of the form "cv array of N pointer to T" into "cv array of N pointer to cv T"

It appears that reinterpret_cast is the only way to perform the conversion.

Suggested resolution:

Artem Livshits proposed a resolution :-

"I suppose if the definition of "similar" pointer types in 7.5 [conv.qual] paragraph 4 was rewritten like this:

> T1 is cv1,0 P0 cv1,1 P1 ... cv1,n-1 Pn-1 cv1,n T
>
> and
>
> T2 is cv1,0 P0 cv1,1 P1 ... cv1,n-1 Pn-1 cv1,n T
>
> where Pi is either a "pointer to" or a "pointer to an array of Ni"; besides P0 may be also a "reference to" or a "reference to an array of N0" (in the case of P0 of T2 being a reference, P0 of T1 may be nothing).

it would address the problem.

In fact I guess Pi in this notation may be also a "pointer to member", so 7.5 [conv.qual]/{4,5,6,7} would be nicely wrapped in one paragraph."

Additional note, February, 2014:

Geoffrey Romer: LWG plans to resolve US 16/LWG 2118, which concerns qualification-conversion of unique_ptr for array types, by effectively punting the issue to core: `unique_ptr<T[]>` will be specified to be convertible to `unique_ptr<U[]>` only if `T(*)[]` is convertible to `U(*)[]`. LWG and LEWG have jointly decided to adopt the same approach for `shared_ptr<T[]>` and `shared_ptr<T[N]>` in the Fundamentals TS. This will probably substantially raise the visibility of core issue 330, which concerns the fact that array types support only top-level qualification conversion of the element type, so it'd be nice if CWG could bump up the priority of that issue.

See also issue 1865.

Proposed resolution (October, 2014):

The resolution is contained in paper N4261.

1816. Unclear specification of bit-field values

Section: 7.8 [conv.integral] **Status:** CD4 **Submitter:** Hubert Tong **Date:** 2013-12-02

[Moved to DR at the November, 2014 meeting.]

7.8 [conv.integral] paragraph 3 says, regarding integral conversions,

> If the destination type is signed, the value is unchanged if it can be represented in the destination type (and bit-field width); otherwise, the value is implementation-defined.

The values that can be represented in a bit-field are not well specified, except for the correspondence with the values of an enumeration in 10.2 [dcl.enum]. In particular, it is not clear whether a bit-field has a sign bit and whether bit-fields may have padding bits.

Another point to note in this wording: paragraph 1 describes the context as

> A prvalue of an integer type can be converted to a prvalue of another integer type.

However, prvalues cannot be bit-fields, so the applicability of the mention of "bit-field width" in paragraph 3 is unclear.

Proposed resolution (February, 2014):

1. Change 7.8 [conv.integral] paragraph 3 as follows:

   > If the destination type is signed, the value is unchanged if it can be represented in the destination type ~~(and bit-field width)~~; otherwise, the value is implementation-defined.

2. Change 8.2.6 [expr.post.incr] paragraph 1 as follows:

   > ...The result is a prvalue. The type of the result is the cv-unqualified version of the type of the operand. **If the operand is a bit-field that cannot represent the incremented value, the resulting value of the bit-field is implementation-defined.** See also 8.7 [expr.add] and 8.18 [expr.ass].

3. Change 8.18 [expr.ass] paragraph 6 as follows:

   > When the left operand of an assignment operator ~~denotes a reference to *T*, the operation assigns to the object of type *T* denoted by the reference~~ **is a bit-field that cannot represent the value of the expression, the resulting value of the bit-field is implementation-defined.**.

4. Change the final bullet of 11.6 [dcl.init] paragraph 17 as follows:

> The semantics of initializers are as follows...
>
> - ...
> - ...no user-defined conversions are considered. If the conversion cannot be done, the initialization is ill-formed. **When initializing a bit-field with a value that it cannot represent, the resulting value of the bit-field is implementation-defined.** [*Note:* An expression of type...

---

238. Precision and accuracy constraints on floating point

Section: 8 [expr] **Status:** CD4 **Submitter:** Christophe de Dinechin **Date:** 31 Jul 2000

[Adopted at the February, 2016 meeting.]

It is not clear what constraints are placed on a floating point implementation by the wording of the Standard. For instance, is an implementation permitted to generate a "fused multiply-add" instruction if the result would be different from what would be obtained by performing the operations separately? To what extent does the "as-if" rule allow the kinds of optimizations (e.g., loop unrolling) performed by FORTRAN compilers?

Proposed resolution (September, 2015):

Change 6.9.1 [basic.fundamental] paragraph 8 as follows:

> There are three *floating point* types: `float`, `double`, and `long double`. The type `double` provides at least as much precision as `float`, and the type `long double` provides at least as much precision as `double`. The set of values of the type `float` is a subset of the set of values of the type `double`; the set of values of the type `double` is a subset of the set of values of the type `long double`. The value representation of floating-point types is implementation-defined. **[*Note:* This International Standard imposes no requirements on the accuracy of floating-point operations; see also _N4606_.18.3.2 [limits]. —*end note*]** *Integral* and *floating* types are collectively called *arithmetic* types. Specializations of the standard **library** template `std::numeric_limits` (21.3 [support.limits]) shall specify the maximum and minimum values of each arithmetic type for an implementation.

1722. Should lambda to function pointer conversion function be `noexcept`?

Section: 8.1.5 [expr.prim.lambda] **Status:** CD4 **Submitter:** Ville Voutilainen **Date:** 2013-07-31

[Moved to DR at the October, 2015 meeting.]

According to 8.1.5 [expr.prim.lambda] paragraph 6,

> The closure type for a non-generic *lambda-expression* with no *lambda-capture* has a public non-virtual non-explicit const conversion function to pointer to function with C++ language linkage (10.5 [dcl.link]) having the same parameter and return types as the closure type's function call operator.

This does not specify whether the conversion function is `noexcept(true)` or `noexcept(false)`. It might be helpful to nail that down.

Proposed resolution (May, 2015):

Change 8.1.5 [expr.prim.lambda] paragraph 6 as follows:

> The closure type for a non-generic *lambda-expression* with no *lambda-capture* has a ~~public non-virtual non-explicit const~~ conversion function to pointer to function with C++ language linkage (10.5 [dcl.link]) having the same parameter and return types as the closure type's function call operator. The value returned by this conversion function shall be the address of a function that, when invoked, has the same effect as invoking the closure type's function call operator. For a generic lambda with no *lambda-capture*, the closure type has a ~~public non-virtual non-explicit const~~ conversion function template to pointer to function. The conversion function template... [*Example:*
>
> ```
> auto GL = [](auto a) { std::cout << a; return a; };
> int (*GL_int)(int) = GL; // OK: through conversion function template
> GL_int(3);               // OK: same as GL(3)
> ```
>
> —*end example*] **The conversion function or conversion function template is public, non-virtual, non-explicit, const, and has a non-throwing exception specification (18.4 [except.spec]).**

1780. Explicit instantiation/specialization of generic lambda `operator()`

Section: 8.1.5 [expr.prim.lambda] **Status:** CD4 **Submitter:** Hubert Tong **Date:** 2013-09-26

[Moved to DR at the November, 2014 meeting.]

Similarly to issue 1738, it is not clear whether it is permitted to explicitly instantiate or specialize the call operator of a polymorphic lambda (via `decltype`).

Proposed resolution (February, 2014):

Add the following as a new paragraph following 8.1.5 [expr.prim.lambda] paragraph 21:

> The closure type associated with a *lambda-expression* has an implicitly-declared destructor (15.4 [class.dtor]).
>
> **A member of a closure type shall not be explicitly instantiated (17.8.1 [temp.inst]), explicitly specialized (17.8.2 [temp.explicit]), or named in a `friend` declaration (14.3 [class.friend]).**

1891. Move constructor/assignment for closure class

Section: 8.1.5 [expr.prim.lambda] **Status:** CD4 **Submitter:** Jonathan Caves **Date:** 2014-03-10

[Moved to DR at the November, 2014 meeting.]

According to 8.1.5 [expr.prim.lambda] paragraph 20,

> The closure type associated with a *lambda-expression* has a deleted (11.4.3 [dcl.fct.def.delete]) default constructor and a deleted copy assignment operator. It has an implicitly-declared copy constructor (15.8 [class.copy]) and may have an implicitly-declared move constructor (15.8 [class.copy]).

However, according to 15.8 [class.copy] paragraph 9,

> If the definition of a class `X` does not explicitly declare a move constructor, one will be implicitly declared as defaulted if and only if
>
> - `X` does not have a user-declared copy constructor,
> - `X` does not have a user-declared copy assignment operator,
> - `X` does not have a user-declared move assignment operator, and
> - `X` does not have a user-declared destructor.

It is not clear how this applies to the closure class. Would it be better to state that the closure class has a defaulted move constructor and a defaulted move assignment operator? There is already wording that handles the case if they are ultimately defined as deleted.

Proposed resolution (October, 2014):

Change 8.1.5 [expr.prim.lambda] paragraph 20 as follows:

> The closure type associated with a *lambda-expression* has ~~a deleted (11.4.3 [dcl.fct.def.delete])~~ **no** default constructor and a deleted copy assignment operator. It has ~~an implicitly-declared~~ **a defaulted** copy constructor ~~(15.8 [class.copy]) and may have an implicitly-declared~~ **and a defaulted** move constructor (15.8 [class.copy]). [*Note:* ~~The copy/move constructor is implicitly defined in the same way as any other implicitly declared copy/move constructor would be implicitly defined~~ **These special member functions are implicitly defined as usual, and might therefore be defined as deleted**. —*end note*]

1942. Incorrect reference to *trailing-return-type*

Section: 8.1.5 [expr.prim.lambda] **Status:** CD4 **Submitter:** Mike Miller **Date:** 2014-06-16

[Moved to DR at the May, 2015 meeting.]

According to 8.1.5 [expr.prim.lambda] paragraph 4,

> If a *lambda-expression* does not include a *lambda-declarator*, it is as if the *lambda-declarator* were `()`. The lambda return type is `auto`, which is replaced by the *trailing-return-type* if provided...

trailing-return-type is a syntactic nonterminal that includes the `->` and thus cannot be used directly to refer to the type. It should instead say something like, ...the type specified by the *trailing-return-type*.

The reference in 11.3.5 [dcl.fct] paragraph 2, "...returning *trailing-return-type*" should be similarly adjusted.

Proposed resolution (November, 2014):

1. Change 8.1.5 [expr.prim.lambda] paragraph 4 as follows:

> If a *lambda-expression* does not include a *lambda-declarator*, it is as if the *lambda-declarator* were `()`. The lambda return type is `auto`, which is replaced by **the type specified by** the *trailing-return-type* if provided and/or deduced from `return`

statements as described in 10.1.7.4 [dcl.spec.auto]. [*Example:*...

2. Change 11.3.5 [dcl.fct] paragraph 2 as follows:

> The type of the *declarator-id* in `D` is "*derived-declarator-type-list* function of (*parameter-declaration-clause*) *cv-qualifier-seq*$_{opt}$ *ref-qualifier*$_{opt}$ returning ~~*trailing-return-type*~~ `U`" **, where `U` is the type specified by the *trailing-return-type***. The optional *attribute-specifier-seq*...

2095. Capturing rvalue references to functions by copy

Section: 8.1.5 [expr.prim.lambda] **Status:** CD4 **Submitter:** Hubert Tong **Date:** 2015-03-07

[Adopted at the February, 2016 meeting.]

According to 8.1.5 [expr.prim.lambda] paragraph 15,

> An entity is captured by copy if it is implicitly captured and the *capture-default* is `=` or if it is explicitly captured with a capture that is not of the form & *identifier* or & *identifier initializer*. For each entity captured by copy, an unnamed non-static data member is declared in the closure type. The declaration order of these members is unspecified. The type of such a data member is the type of the corresponding captured entity if the entity is not a reference to an object, or the referenced type otherwise.

It's not clear how to handle capture by copy when the entity is an rvalue reference to function. In particular, this appears to be a contradiction with 8.1.5 [expr.prim.lambda] paragraph 3,

> An implementation shall not add members of rvalue reference type to the closure type.

Proposed resolution (September, 2015):

Change 8.1.5 [expr.prim.lambda] paragraph 15 as follows:

> An entity is *captured by copy* if it is implicitly captured and the *capture-default* is `=` or if it is explicitly captured with a capture that is not of the form `&` *identifier* or `&` *identifier initializer*. For each entity captured by copy, an unnamed non-static data member is declared in the closure type. The declaration order of these members is unspecified. The type of such a data member is **the referenced type if the entity is a reference to an object, an lvalue reference to the referenced function type if the entity is a reference to a function, or** the type of the corresponding captured entity ~~if the entity is not a reference to an object, or the referenced type~~ otherwise. ~~[*Note:* If the captured entity is a reference to a function, the corresponding data member is also a reference to a function. — *end note*]~~ A member of an anonymous union shall not be captured by copy.

1885. Return value of a function is underspecified

Section: 8.2.2 [expr.call] **Status:** CD4 **Submitter:** Jens Maurer **Date:** 2014-02-28

[Moved to DR at the November, 2014 meeting.]

The intent is that a function call is a temporary expression whose result is a temporary, but that appears not to be said anywhere. It should also be clarified that a `return` statement in a function with a class return type copy-initializes the temporary that is the result. The sequencing of the initialization of the returned temporary, destruction of temporaries in the `return` expression, and destruction of automatic variables should be make explicit.

Proposed resolution (October, 2014):

Change 9.6.3 [stmt.return] paragraphs 2-3 as follows:

> ~~A return statement with neither an *expression* nor a *braced-init-list* can be used only in functions that do not return a value, that is,~~ **The *expression* or *braced-init-list* of a return statement is called its operand. A return statement with no operand shall be used only in** a function ~~with the~~ **whose** return type **is** *cv* `void`, a constructor (15.1 [class.ctor]), or a destructor (15.4 [class.dtor]). **A return statement with an operand of type `void` shall be used only in a function whose return type is *cv* `void`.** A return statement with ~~an expression of non-void type can be used only~~ **any other operand shall be used only** in ~~functions returning a value; the value of the expression is returned to the caller of the function. The value of the expression is implicitly converted to the return type of the function in which it appears~~ **a function whose return type is not *cv* `void`; the return statement initializes the object or reference to be returned by copy-initialization (11.6 [dcl.init]) from the operand**. **[*Note:*** A return statement can involve the construction and copy or move of a temporary object (15.2 [class.temporary]). ~~[*Note:*~~ A copy or move operation associated with a return statement may be elided or considered as an rvalue for the purpose of overload resolution in selecting a constructor (15.8 [class.copy]). — *end note*] ~~A return statement with a *braced-init-list* initializes the object or reference to be returned from the function by copy-list-initialization (11.6.4 [dcl.init.list]) from the specified initializer list.~~ [*Example:*

```
std::pair<std::string,int> f(const char* p, int x) {
  return {p,x};
}
```

—*end example*] Flowing off the end of a function is equivalent to a return with no value; this results in undefined behavior in a value-returning function.

~~A return statement with an expression of type `void` can be used only in functions with a return type of *cv* `void`; the expression is evaluated just before the function returns to its caller.~~ **The copy-initialization of the returned entity is sequenced before the destruction of temporaries at the end of the full-expression established by the operand of the return statement, which, in turn, is sequenced before the destruction of local variables (9.6 [stmt.jump]) of the block enclosing the return statement.**

(See also the related changes in the resolution of issue 1299.)

2176. Destroying the returned object when a destructor throws

Section: 8.2.2 [expr.call] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2015-09-28

[Adopted at the February, 2016 meeting.]

Consider the following example:

```
#include <stdio.h>

struct X {
  X() { puts("X()"); }
  X(const X&) { puts("X(const X&)"); }
  ~X() { puts("~X()"); }
};

struct Y { ~Y() noexcept(false) { throw 0; } };

X f() {
  try {
    Y y;
    return {};
  } catch (...) {
  }
  return {};
}

int main() {
  f();
}
```

Current implementations print `X()` twice but `~X()` only once. That is obviously wrong, but it is not clear that the current wording covers this case.

Proposed resolution (February, 2016):

Change 18.2 [except.ctor] paragraph 2 as follows:

> The destructor is invoked for each automatic object of class type constructed**, but not yet destroyed,** since the try block was entered. **If an exception is thrown during the destruction of temporaries or local variables for a `return` statement (9.6.3 [stmt.return]), the destructor for the returned object (if any) is also invoked.** The ~~automatic~~ objects are destroyed in the reverse order of the completion of their construction. **[*Example:*
>
> ```
> struct A { };
>
> struct Y { ~Y() noexcept(false) { throw 0; } };
>
> A f() {
>   try {
>     A a;
>     Y y;
>     A b;
>     return {};   // #1
>   } catch (...) {
>   }
>   return {};     // #2
> }
> ```
>
> **At `#1`, the returned object of type `A` is constructed. Then, the local variable `b` is destroyed (9.6 [stmt.jump]). Next, the local variable `y` is destroyed, causing stack unwinding, resulting in the destruction of the returned object, followed by the destruction of the local variable `a`. Finally, the returned object is constructed again at `#2`. —*end example*]**

1920. Qualification mismatch in *pseudo-destructor-name*

Section: 8.2.4 [expr.pseudo] **Status:** CD4 **Submitter:** David Majnemer **Date:** 2014-05-01

[Moved to DR at the May, 2015 meeting.]

An example like

```
typedef int T;
typedef const T CT;

void blah2(T *a) {
 a->CT::~T();
}
```

is ill-formed, because 8.2.4 [expr.pseudo] paragraph 2 requires that the two *type-name*s in the *qualified-id* be the same type. The corresponding case for a real destructor, however, is allowed because of the provision in 12.1 [class.name] paragraph 5 ignoring cv-qualifiers in a *typedef-name* referring to a class type. The specification for pseudo-destructors should be adjusted accordingly.

Proposed resolution (November, 2014):

Change 8.2.4 [expr.pseudo] paragraph 2 as follows:

> ...The cv-unqualified versions of the object type and of the type designated by the *pseudo-destructor-name* shall be the same type. Furthermore, the two *type-name*s in a *pseudo-destructor-name* of the form
>
> *nested-name-specifier*$_{opt}$ *type-name* `::` `~` *type-name*
>
> shall designate the same scalar type **(ignoring cv-qualification)**.

1832. Casting to incomplete enumeration

Section: 8.2.9 [expr.static.cast] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-01-16

[Moved to DR at the November, 2014 meeting.]

The specification of casting to an enumeration type in 8.2.9 [expr.static.cast] paragraph 10 does not require that the enumeration type be complete. Should it? (There is variation among implementations.)

Proposed resolution (February, 2014):

Change 8.2.9 [expr.static.cast] paragraph 10 as follows:

> A value of integral or enumeration type can be explicitly converted to ~~an~~ **a complete** enumeration type. The value is...

1800. Pointer to member of nested anonymous union

Section: 8.3.1 [expr.unary.op] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2013-10-22

[Moved to DR at the November, 2014 meeting.]

According to 8.3.1 [expr.unary.op] paragraph 3,

> The result of the unary `&` operator is a pointer to its operand. The operand shall be an lvalue or a *qualified-id*. If the operand is a *qualified-id* naming a non-static member `m` of some class `C` with type `T`, the result has type "pointer to member of class `C` of type `T`" and is a prvalue designating `C::m`.

It is not clear whether this wording applies to variant members of `C` (i.e., members of nested anonymous unions) or only to its non-variant members. For example, given

```
struct A { union { int n; }; };
auto x = &A::n;
```

should the type of `x` be `int A::*` or `int A::`*anon*`::*`? Current implementations choose the former.

Proposed resolution (February, 2014):

Change 8.3.1 [expr.unary.op] paragraph 3 as follows:

> The result of the unary `&` operator is a pointer to its operand. The operand shall be an lvalue or a *qualified-id*. If the operand is a *qualified-id* naming a non-static **or variant** member `m` of some class `C` with type `T`, the result has type "pointer to member of class `C` of type `T`" and is a prvalue designating `C::m`. Otherwise...

1971. Unclear disambiguation of destructor and `operator~`

Section: 8.3.1 [expr.unary.op] **Status:** CD4 **Submitter:** Hubert Tong **Date:** 2014-07-15

[Moved to DR at the May, 2015 meeting.]

There is language in 8.3.1 [expr.unary.op] paragraph 10 to disambiguate destructor references from references to `operator~`:

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

However, it is not clear whether this is intended to apply to an example like,

```
struct X {
  X(int);
  operator int();
  void foo() {
    ~X(0);
  }
};
```

where the form of reference has an apparent argument.

Proposed resolution (November, 2014):

Change 8.3.1 [expr.unary.op] paragraph 10 as follows:

> The operand of `~` shall have integral or unscoped enumeration type; the result is the one's complement of its operand. Integral promotions are performed. The type of the result is the type of the promoted operand. There is an ambiguity ~~in the *unary-expression* `~X()`, where `X` is~~ **in the grammar when `~` is followed by** a *class-name* or *decltype-specifier*. The ambiguity is resolved ~~in favor of~~ **by** treating `~` as ~~a~~ **the** unary complement **operator** rather than ~~treating `~X` as referring to~~ **as the start of an *unqualified-id* naming** a destructor. **[*Note:* Because the grammar does not permit an operator to follow the `.`, `->`, or `::` tokens, a `~` followed by a *class-name* or *decltype-specifier* in a member access expression or *qualified-id* is unambiguously parsed as a destructor name. —*end note*]**

1653. Removing deprecated increment of `bool`

Section: 8.3.2 [expr.pre.incr] **Status:** CD4 **Submitter:** Bjarne Stroustrup **Date:** 2013-04-19

[Adopted at the October, 2015 meeting as P0002R1.]

The preincrement (8.3.2 [expr.pre.incr]) and postincrement (8.2.6 [expr.post.incr]) operators can be applied to operands of type `bool`, setting the operand to `true`, but this use is deprecated. Can it now be removed altogether?

1748. Placement new with a null pointer

Section: 8.3.4 [expr.new] **Status:** CD4 **Submitter:** Marc Glisse **Date:** 2013-09-11

[Moved to DR at the November, 2014 meeting.]

According to 8.3.4 [expr.new] paragraph 15,

> [*Note:* unless an allocation function is declared with a non-throwing *exception-specification* (18.4 [except.spec]), it indicates failure to allocate storage by throwing a `std::bad_alloc` exception (Clause 18 [except], 21.6.3.1 [bad.alloc]); it returns a non-null pointer otherwise. If the allocation function is declared with a non-throwing *exception-specification*, it returns null to indicate failure to allocate storage and a non-null pointer otherwise. —*end note*] If the allocation function returns null, initialization shall not be done, the deallocation function shall not be called, and the value of the *new-expression* shall be null.

This wording applies even to the non-replaceable placement forms defined in 21.6.2.3 [new.delete.placement] that simply return the supplied pointer as the result of the allocation function. Compilers are thus required to check for a null pointer and avoid the initialization if one is used. This test is unnecessary overhead; it should be the user's responsibility to ensure that a null pointer is not used in these forms of placement new, just as for other cases when a pointer is dereferenced.

Proposed resolution (February, 2014):

Change 8.3.4 [expr.new] paragraph 15 as follows:

> [*Note:* unless an allocation function is declared with a non-throwing *exception-specification* (18.4 [except.spec]), it indicates failure to allocate storage by throwing a `std::bad_alloc` exception (Clause 18 [except], 21.6.3.1 [bad.alloc]); it returns a non-null pointer otherwise. If the allocation function is declared with a non-throwing *exception-specification*, it returns null to indicate failure to allocate storage and a non-null pointer otherwise. —*end note*] If **the allocation function is a reserved placement allocation function (21.6.2.3 [new.delete.placement]) that returns null, the behavior is undefined. Otherwise, if** the allocation function returns null, initialization shall not be done, the deallocation function shall not be called, and the value of the *new-expression* shall be null.

1851. `decltype(auto)` in *new-expression*s

Section: 8.3.4 [expr.new] **Status:** CD4 **Submitter:** Daveed Vandevoorde **Date:** 2014-02-04

[Moved to DR at the November, 2014 meeting.]

10.1.7.4 [dcl.spec.auto] paragraph 5 says that a placeholder type (presumably including `decltype(auto)`) can appear in a *new-expression*. However, 8.3.4 [expr.new] mentions only `auto`, not `decltype(auto)`.

Proposed resolution (February, 2014):

Change 8.3.4 [expr.new] paragraph 2 as follows:

> If the ~~`auto` *type-specifier*~~ **a placeholder type (10.1.7.4 [dcl.spec.auto])** appears in the *type-specifier-seq* of a *new-type-id* or *type-id* of a *new-expression*, the *new-expression* shall contain...

1992. `new (std::nothrow) int[N]` **can throw**

Section: 8.3.4 [expr.new] **Status:** CD4 **Submitter:** Martin Sebor **Date:** 2014-08-27

[Adopted at the February, 2016 meeting.]

According to 8.3.4 [expr.new] paragraph 7,

> If the expression, after converting to `std::size_t`, is a core constant expression and the expression is erroneous, the program is ill-formed. Otherwise, a *new-expression* with an erroneous expression does not call an allocation function and terminates by throwing an exception of a type that would match a handler (18.3 [except.handle]) of type `std::bad_array_new_length` (21.6.3.2 [new.badlength]).
>
> This wording makes no provision for an expression like
>
> ```
> new (std::nothrow) int[N]
> ```
>
> which most programmers would intuitively expect not to throw an exception under any condition.

Proposed resolution (May, 2015) [SUPERSEDED]:

Change the last part of 8.3.4 [expr.new] paragraph 7 as follows, converting the running text into bullets, and making the last sentence into a paragraph 8:

> ...If the expression~~,~~ **is erroneous** after converting to `std::size_t`~~,~~**:**
>
> - **if the *expression*** is a core constant expression ~~and the expression is erroneous~~, the program is ill-formed~~.~~**;**
> - ~~Otherwise~~ **otherwise**, ~~a *new-expression* with an erroneous expression does not call~~ an allocation function **is not called; instead**
>   - **if the allocation function that would have been called is non-throwing (18.4 [except.spec]), the value of the *new-expression* is the null pointer value of the required result type;**
>   - ~~and~~ **otherwise, the *new-expression*** terminates by throwing an exception of a type that would match a handler (18.3 [except.handle]) of type `std::bad_array_new_length` (21.6.3.2 [new.badlength]).
>
> When the value of the *expression* is zero, the allocation function is called to allocate an array with no elements.

Notes from the October, 2015 meeting:

The text in 15.4 paragraph 15 should also be changed.

Proposed resolution (January, 2016):

1. Change 8.3.4 [expr.new] paragraph 7 as follows, dividing the running text into bullets and making the last sentence into a new paragraph:

   > The expression in a *noptr-new-declarator* is erroneous if:
   >
   > - ...
   >
   > If the *expression*~~,~~ **is erroneous** after converting to `std::size_t`~~,~~**:**
   >
   > - **if the *expression*** is a core constant expression ~~and the expression is erroneous~~, the program is ill-formed~~.~~**;**
   > - ~~Otherwise~~ **otherwise**, ~~a *new-expression* with an erroneous expression does not call~~ an allocation function ~~and~~ **is not called; instead**
   >   - **if the allocation function that would have been called has a non-throwing exception specification (18.4 [except.spec]), the value of the *new-expression* is the null pointer value of the required result type;**
   >   - **otherwise, the *new-expression*** terminates by throwing an exception of a type that would match a handler (18.3 [except.handle]) of type `std::bad_array_new_length` (21.6.3.2 [new.badlength]).
   >
   > When the value of the expression is zero, the allocation function is called to allocate an array with no elements.

2. Change 18.4 [except.spec] paragraph 14 as follows:

> The *set of potential exceptions of an expression* `e` is empty if e is a core constant expression (8.20 [expr.const]). Otherwise, it is the union of the sets of potential exceptions of the immediate subexpressions of `e`, including default argument expressions used in a function call, combined with a set *S* defined by the form of e, as follows:
>
> - ...
> - If `e` implicitly invokes ~~a~~ **one or more** function**s** (such as an overloaded operator, an allocation function in a *new-expression*, or a destructor if `e` is a full-expression (4.6 [intro.execution])), *S* is the ~~set of potential exceptions of the function.~~ **union of:**
>   - **the sets of potential exceptions of all such functions, and**
>   - **if `e` is a *new-expression* with a non-constant *expression* in the *noptr-new-declarator* (8.3.4 [expr.new]) and the allocation function selected for `e` has a non-empty set of potential exceptions, the set containing `std::bad_array_new_length`.**
> - ...
> - ~~If `e` is a *new-expression* with a non-constant *expression* in the *noptr-new-declarator* (8.3.4 [expr.new]), *S* consists of the type `std::bad_array_new_length`.~~
>
> [*Example:*...

3. Change the example in 18.4 [except.spec] bullet 17.2 as follows:

```
  struct A {
    A(int = (A(5), 0)) noexcept;
    A(const A&) throw();
    A(A&&) throw();
    ~A() throw(X);
  };
  struct B {
    B() throw();
    B(const B&) = default; // exception specification contains no types
    B(B&&, int = (throw Y(), 0)) noexcept;
    ~B() throw(Y);
  };
  int n = 7;
  struct D : public A, public B {
    int * p = new (std::nothrow) int[n];
  // exception specification of D::D() contains X and std::bad_array_new_length
    // exception specification of D::D(const D&) contains no types
    // exception specification of D::D(D&&) contains Y
    // exception specification of D::~D() contains X and Y
  };
  struct exp : std::bad_alloc {};
  void *operator new[](size_t) throws(exp);
  struct E : public A {
    int * p = new int[n];
    // exception specification of E::E() contains X, exp, and std::bad_array_new_length
};
```

2130. Over-aligned types in *new-expression*s

Section: 8.3.4 [expr.new] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2015-05-28

[Adopted at the February, 2016 meeting.]

According to 8.3.4 [expr.new] paragraph 1,

> It is implementation-defined whether over-aligned types are supported (6.11 [basic.align]).

However, there is no mechanism for informing an allocation function of the required alignment for over-aligned types. Nevertheless, 6.11 [basic.align] paragraph 9 says:

> Additionally, a request for runtime allocation of dynamic storage for which the requested alignment cannot be honored shall be treated as an allocation failure.

This seems contradictory.

Proposed resolution (October, 2015):

Change 6.11 [basic.align] paragraph 9 as follows:

> If a request for a specific extended alignment in a specific context is not supported by an implementation, the program is ill-formed. ~~Additionally, a request for runtime allocation of dynamic storage for which the requested alignment cannot be honored shall be treated as an allocation failure.~~

2141. Ambiguity in *new-expression* with *elaborated-type-specifier*

Section: 8.3.4 [expr.new] **Status:** CD4 **Submitter:** Hubert Tong **Date:** 2015-06-12

[Adopted at the February, 2016 meeting.]

Consider the following example:

```
struct A { };

void foo() {
  new struct A { };
}
```

This could be either an *elaborated-type-specifier* followed by a *braced-init-list* or a *class-specifier*. There does not appear to be a disambiguation rule for this case.

One possibility for addressing this could be to use *trailing-type-specifier-seq* instead of *type-specifier-seq* in *new-type-id*. That could also be a purely syntactic alternative to the resolution of issue 686: change all uses of *type-specifier-seq* to *trailing-type-specifier-seq* and provide a new grammar production for use in *alias-declaration*.

Proposed resolution (February, 2016):

1. Change the grammar in 10 [dcl.dcl] paragraph 1 as follows:

   alias-declaration:
   `using` *identifier attribute-specifier-seq*$_{opt}$ `=` ***defining-type-id***

2. Change 10 [dcl.dcl] paragraph 8 as follows:

   Each *init-declarator* in the *init-declarator-list* contains exactly one *declarator-id*, which is the name declared by that *init-declarator* and hence one of the names declared by the declaration. The ***defining-type-specifier***s (10.1.7 [dcl.type]) in the *decl-specifier-seq* and the recursive declarator structure of the *init-declarator* describe a type (11.3 [dcl.meaning]), which is then associated with the name being declared by the *init-declarator*.

3. Change the grammar in 10.1 [dcl.spec] paragraph 1 as follows:

   decl-specifier:
   storage-class-specifier
   defining-type-specifier
   function-specifier
   `friend`
   `typedef`
   `constexpr`

4. Change 10.1 [dcl.spec] paragraph 3 as follows:

   If a *type-name* is encountered while parsing a *decl-specifier-seq*, it is interpreted as part of the *decl-specifier-seq* if and only if there is no previous ***defining-type-specifier*** other than a *cv-qualifier* in the *decl-specifier-seq*. The sequence...

5. Change 10.1.3 [dcl.typedef] paragraph 1 as follows:

   Declarations containing the *decl-specifier* `typedef` declare identifiers that can be used later for naming fundamental (6.9.1 [basic.fundamental]) or compound (6.9.2 [basic.compound]) types. The `typedef` specifier shall not be combined in a *decl-specifier-seq* with any other kind of specifier except a ***defining-type-specifier***, and it shall not be used in the *decl-specifier-seq* of a *parameter-declaration* (11.3.5 [dcl.fct]) nor in the *decl-specifier-seq* of a *function-definition* (11.4 [dcl.fct.def]).

6. Change 10.1.3 [dcl.typedef] paragraph 2 as follows:

   A *typedef-name* can also be introduced by an *alias-declaration*. The identifier following the `using` keyword becomes a *typedef-name* and the optional *attribute-specifier-seq* following the identifier appertains to that *typedef-name*. ~~It~~ **The *defining-type-specifier-seq* of the *defining-type-id* may define a class or enumeration only if the *alias-declaration* is not the *declaration* of a *template-declaration*. Such a *typedef-name*** has the same semantics as if it were introduced by the `typedef` specifier. In particular, it does not define a new type. [*Example:*

7. Change 10.1.7 [dcl.type] paragraph 1 as follows:

   type-specifier:
   ~~*trailing-type-specifier*~~
   ~~*class-specifier*~~
   ~~*enum-specifier*~~
   ~~*trailing-type-specifier:*~~
   simple-type-specifier
   elaborated-type-specifier
   typename-specifier
   cv-qualifier
   type-specifier-seq:
   type-specifier attribute-specifier-seq$_{opt}$
   type-specifier type-specifier-seq
   defining-type-specifier:

type-specifier
class-specifier
enum-specifier

~~*trailing*~~***defining***-*type-specifier-seq:*
~~*trailing*~~***defining***-*type-specifier attribute-specifier-seq*$_{opt}$
~~*trailing*~~***defining***-*type-specifier* ~~*trailing*~~***defining***-*type-specifier-seq*

The optional *attribute-specifier-seq* in a *type-specifier-seq* or a ~~*trailing*~~***defining***-*type-specifier-seq* appertains to the type denoted by the preceding *type-specifier*s **or *defining-type-specifier*s** (11.3 [dcl.meaning]). The *attribute-specifier-seq* affects the type only for the declaration it appears in, not other declarations involving the same type.

8. Change 10.1.7.2 [dcl.type.simple] paragraph 2 as follows:

As a general rule, at most one ***defining-****type-specifier* is allowed in the complete *decl-specifier-seq* of a declaration or in a *type-specifier-seq* or ~~*trailing*~~***defining***-*type-specifier-seq*. The only exceptions to this rule are the following:...

9. Change 10.1.7 [dcl.type] paragraph 3 as follows:

Except in a declaration of a constructor, destructor, or conversion function, at least one ***defining-****type-specifier* that is not a *cv-qualifier* shall appear in a complete *type-specifier-seq* or a complete *decl-specifier-seq*.[95] ~~*A type-specifier-seq* shall not define a class or enumeration unless it appears in the *type-id* of an *alias-declaration* (10.1.3 [dcl.typedef]) that is not the declaration of a *template-declaration*.~~

10. Change the grammar in 11 [dcl.decl] paragraph 4 as follows:

trailing-return-type:
-> ~~*trailing-type-specifier-seq abstract-declarator*$_{opt}$~~ ***type-id***

11. Change the grammar in 11.1 [dcl.name] paragraph 1 as follows:

type-id:
type-specifier-seq abstract-declarator$_{opt}$
defining-type-id:
defining-type-specifier-seq abstract-declarator$_{opt}$

12. Change 15.3.2 [class.conv.fct] paragraph 1 as follows:

...A *decl-specifier* in the *decl-specifier-seq* of a conversion function (if any) shall be neither a ***defining-****type-specifier* nor static. Type of the conversion function

---

1788. Sized deallocation of array of non-class type

Section: 8.3.5 [expr.delete] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2013-09-29

[Moved to DR at the November, 2014 meeting.]

The changes from N3778 require use of a sized deallocator for a case like

```
char *p = new char[32];
void f() { delete [] p; }
```

That is unimplementable under current ABIs, which do not store the array size for such allocations. It should instead be unspecified or implementation-defined whether the sized form of `operator[]` is used for a pointer to a type other than a class with a non-trivial destructor or array thereof.

Proposed resolution (February, 2014) [SUPERSEDED]:

Change 8.3.5 [expr.delete] paragraph 10 as follows:

If ~~the type is complete and if~~ deallocation function lookup finds both a usual deallocation function with only a pointer parameter and a usual deallocation function with both a pointer parameter and a size parameter, then

- **for the first alternative (delete object), if the type of the object to be deleted is complete, and for the second alternative (delete array), if the type of the object to be deleted is a complete class type with a non-trivial destructor, then** the selected deallocation function shall be the one with two parameters;
- **otherwise, it is implementation-defined which deallocation function is selected.**

  Otherwise, the selected deallocation function shall be the function with one parameter.

Additional note, February, 2014:

It is not clear that this resolution accurately reflects the intent of the issue. In particular, it changes deletion of a pointer to incomplete type from requiring use of the single-parameter version to being implementation-defined. Also, the "type of the object to be deleted" in the array case is always an array type and thus cannot be "a complete class type with a non-trivial destructor." The issue has consequently been returned to "review" status.

Proposed resolution (June, 2014):

Change 8.3.5 [expr.delete] paragraph 10 as follows:

> If ~~the type is complete and if~~ deallocation function lookup finds both a usual deallocation function with only a pointer parameter and a usual deallocation function with both a pointer parameter and a size parameter, ~~then the selected deallocation function shall be the one with two parameters. Otherwise, the selected deallocation function shall be the function with one parameter.~~ **the function to be called is selected as follows:**
>
> - **If the type is complete and if, for the second alternative (delete array) only, the operand is a pointer to a class type with a non-trivial destructor or a (possibly multi-dimensional) array thereof, the function with two parameters is selected.**
> - **Otherwise, it is unspecified which of the two deallocation functions is selected.**

1465. `noexcept` **and** `std::bad_array_new_length`

Section: 8.3.7 [expr.unary.noexcept] **Status:** CD4 **Submitter:** Daniel Krügler **Date:** 2012-02-12

[Moved to DR at the November, 2014 meeting.]

The list of causes for a `false` result of the `noexcept` operator does not include a *new-expression* with a non-constant array bound, which could result in an exception even if the allocation function that would be called is declared not to throw (see 8.3.4 [expr.new] paragraph 7).

Proposed resolution (June, 2012):

This issue is resolved by the resolution of issue 1351.

---

242. Interpretation of old-style casts

Section: 8.4 [expr.cast] **Status:** CD4 **Submitter:** Mike Miller **Date:** 30 Aug 2000

[Adopted at the February, 2016 meeting.]

The meaning of an old-style cast is described in terms of `const_cast`, `static_cast`, and `reinterpret_cast` in 8.4 [expr.cast] paragraph 5. Ignoring `const_cast` for the moment, it basically says that if the conversion performed by a given old-style cast is one of those performed by `static_cast`, the conversion is interpreted as if it were a `static_cast`; otherwise, it's interpreted as if it were a `reinterpret_cast`, if possible. The following example is given in illustration:

```
struct A {};
struct I1 : A {};
struct I2 : A {};
struct D : I1, I2 {};
A *foo( D *p ) {
    return (A*)( p ); // ill-formed static_cast interpretation
}
```

The obvious intent here is that a derived-to-base pointer conversion is one of the conversions that can be performed using `static_cast`, so `(A*)(p)` is equivalent to `static_cast<A*>(p)`, which is ill-formed because of the ambiguity.

Unfortunately, the description of `static_cast` in 8.2.9 [expr.static.cast] does NOT support this interpretation. The problem is in the way 8.2.9 [expr.static.cast] lists the kinds of casts that can be performed using `static_cast`. Rather than saying something like "All standard conversions can be performed using `static_cast`," it says

> An expression e can be explicitly converted to a type `T` using a `static_cast` of the form `static_cast<T>(e)` if the declaration "`T t(e);`" is well-formed, for some invented temporary variable `t`.

Given the declarations above, the hypothetical declaration

```
A* t(p);
```

is NOT well-formed, because of the ambiguity. Therefore the old-style cast `(A*)(p)` is NOT one of the conversions that can be performed using `static_cast`, and `(A*)(p)` is equivalent to `reinterpret_cast<A*>(p)`, which is well-formed under 8.2.10 [expr.reinterpret.cast] paragraph 7.

Other situations besides ambiguity which might raise similar questions include access violations, casting from virtual base to derived, and casting pointers-to-members when virtual inheritance is involved.

Proposed resolution (October, 2015):

1. Change 8.2.9 [expr.static.cast] paragraph 2 as follows:

> An lvalue of type "*cv1* `B`", where `B` is a class type, can be cast to type "reference to *cv2* `D`", where `D` is a class derived (Clause 13 [class.derived]) from `B`, if ~~a valid standard conversion from "pointer to `D`" to "pointer to `B`" exists (7.11 [conv.ptr]), *cv2*~~ is the same cv-qualification as, or greater cv-qualification than, *cv1*~~, and~~. **If** `B` is ~~neither~~ a virtual base class

of D ~~nor~~ **or** a base class of a virtual base class of D**, or if no valid standard conversion from "pointer to** D**" to "pointer to** B**" exists (7.11 [conv.ptr]), the program is ill-formed**. ~~The result has type "*cv2* D".~~ An xvalue of type "*cv1* B" ~~may~~ **can** be cast to type "rvalue reference to *cv2* D" with the same constraints as for an lvalue of type "*cv1* B". If the object of type "*cv1* B" is actually a subobject of an object of type D, the result refers to the enclosing object of type D. Otherwise, the behavior is undefined. [*Example:*...

2. Change 8.2.9 [expr.static.cast] paragraph 4 as follows:

> An expression e can be explicitly converted to a type T ~~using a `static_cast` of the form `static_cast<T>(e)` if the declaration `T t(e);` is well-formed, for some invented temporary variable `t` (11.6 [dcl.init])~~ **there is an implicit conversion sequence (16.3.3.1 [over.best.ics]) from** e **to** T**, or if overload resolution for a direct-initialization (11.6 [dcl.init]) of an object or reference of type** T **from** e **would find at least one viable function (16.3.2 [over.match.viable])**. The effect of such an explicit conversion is the same as performing the declaration and initialization
>
> ```
> T t(e);
> ```
>
> **for some invented temporary variable** t **(11.6 [dcl.init])** and then using the temporary variable as the result of the conversion. **[*Note:* The conversion is ill-formed when attempting to convert an expression of class type to an inaccessible or ambiguous base class. —*end note*]** The expression e is used as a glvalue if and only if the initialization uses it as a glvalue.

3. Change 8.2.9 [expr.static.cast] paragraph 11 as follows:

> A prvalue of type "pointer to *cv1* B", where B is a class type, can be converted to a prvalue of type "pointer to *cv2* D", where D is a class derived (Clause 13 [class.derived]) from B, if ~~a valid standard conversion from "pointer to D" to "pointer to B" exists (7.11 [conv.ptr]),~~ *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*~~, and~~**. If** B is ~~neither~~ a virtual base class of D ~~nor~~ **or** a base class of a virtual base class of D**, or if no valid standard conversion from "pointer to** D**" to "pointer to** B**" exists (7.11 [conv.ptr]), the program is ill-formed**. The null pointer value (7.11 [conv.ptr]) is converted to the null pointer value of the destination type. If the prvalue of type "pointer to *cv1* B" points to a B that is actually a subobject of an object of type D, the resulting pointer points to the enclosing object of type D. Otherwise, the behavior is undefined.

4. Change 8.2.9 [expr.static.cast] paragraph 12 as follows:

> A prvalue of type "pointer to member of D of type *cv1* T" can be converted to a prvalue of type "pointer to member of B~~"~~ of type *cv2* T**"**, where B is a base class (Clause 13 [class.derived]) of D, if ~~a valid standard conversion from "pointer to member of B of type T" to "pointer to member of D of type T" exists (7.12 [conv.mem]), and~~ *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*.[70] **If no valid standard conversion from "pointer to member of** B **of type** T**" to "pointer to member of** D **of type** T**" exists (7.12 [conv.mem]), the program is ill-formed.** The null member pointer value (7.12 [conv.mem]) is converted to the null member pointer value of the destination type. If class B contains the original member, or is a base or derived class of the class containing the original member, the resulting pointer to member points to the original member. Otherwise, the behavior is undefined. [*Note:* although class B need not contain the original member, the dynamic type of the object with which indirection through the pointer to member is performed must contain the original member; see 8.5 [expr.mptr.oper]. —*end note*]

1865. Pointer arithmetic and multi-level qualification conversions

Section: 8.7 [expr.add] **Status:** CD4 **Submitter:** Geoffrey Romer **Date:** 2014-02-15

[Moved to DR at the November, 2014 meeting.]

The resolution of issue 1504 added 8.7 [expr.add] paragraph 7:

> For addition or subtraction, if the expressions P or Q have type "pointer to *cv* T", where T is different from the cv-unqualified array element type, the behavior is undefined.

This wording was intended to address derived-base conversion in pointer arithmetic, but it inadvertently categorized as undefined behavior previously well-defined pointer arithmetic on pointers that are the result of multi-level qualification conversions. For example:

```
void f() {
  int i = 0;
  int* arr[3] = {&i, &i, &i};
  int const * const * aptr = arr;
  assert(aptr[2] == &i);
}
```

This now has undefined behavior because the type of `*aptr` is "pointer to `const int`," which is different from the cv-unqualified array element type, "pointer to `int`."

See also issue 330.

Proposed Resolution (July, 2014):

Change 8.7 [expr.add] paragraph 7 as follows:

> For addition or subtraction, if the expressions P or Q have type "pointer to *cv* T", where T ~~is different from the cv-unqualified~~ **and the** array element type **are not similar (7.5 [conv.qual])**, the behavior is undefined. [*Note:* In particular, a pointer to a base

class cannot be used for pointer arithmetic when the array contains objects of a derived class type. —*end note*]

1596. Non-array objects as `array[1]`

Section: 8.9 [expr.rel] **Status:** CD4 **Submitter:** Daniel Krügler **Date:** 2012-12-20

[Moved to DR at the November, 2014 meeting.]

The provision to treat non-array objects as if they were array objects with a bound of 1 is given only for pointer arithmetic in C++ (8.7 [expr.add] paragraph 4). C99 supplies similar wording for the relational and equality operators, explicitly allowing pointers resulting from such implicit-array treatment to be compared. C++ should follow suit.

Proposed resolution (August, 2013):

1. Change 8.3.1 [expr.unary.op] paragraph 3 as follows:

> ...Otherwise, if the type of the expression is `T`, the result has type "pointer to `T`" and is a prvalue that is the address of the designated object (4.4 [intro.memory]) or a pointer to the designated function. [*Note:* In particular, the address of an object of type "*cv* `T`" is "pointer to *cv* `T`", with the same cv-qualification. —*end note*] **For purposes of pointer arithmetic (8.7 [expr.add]) and comparison (8.9 [expr.rel], 8.10 [expr.eq]), an object that is not an array element whose address is taken in this way is considered to belong to an array with one element of type `T`.** [*Example:*
>
> ```
> struct A { int i; };
> struct B : A { };
> ... &B::i ...        // has type int A::*
> int a;
> int* p1 = &a;
> int* p2 = p1 + 1;    // Defined behavior
> bool b = p2 > p1;    // Defined behavior, with value true
> ```
>
> —*end example*] [*Note:* a pointer to member...

2. Delete 8.7 [expr.add] paragraph 4:

> ~~For the purposes of these operators, a pointer to a nonarray object behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.~~

3. Change 8.7 [expr.add] paragraph 5 as follows:

> When an expression that has integral type is added to or subtracted from a pointer, the result has the type of the pointer operand. If the pointer operand points to an element of an array object **[*Footnote:* An object that is not an array element is considered to belong to a single-element array for this purpose; see 8.3.1 [expr.unary.op] —*end footnote*]**, and the array is large enough, the result points to an element...

4. Change 8.9 [expr.rel] paragraph 3 as follows:

> Comparing pointers to objects **[*Footnote:* An object that is not an array element is considered to belong to a single-element array for this purpose; see 8.3.1 [expr.unary.op] —*end footnote*]** is defined as follows:...

[Drafting note: No change is proposed for 8.10 [expr.eq], since the comparison is phrased in terms of "same address", not in terms of array elements, so the handling of one-past-the-end addresses falls out of the specification of pointer arithmetic.]

1652. Object addresses in `constexpr` expressions

Section: 8.10 [expr.eq] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2013-04-15

[Moved to DR at the May, 2015 meeting.]

Pointer equality is defined by reference to the addresses of the objects designated by the pointer values, reflecting the implementation technique of most/all compilers. However, this definition is intrinsically a runtime property, and such a description is inappropriate with respect to `constexpr` expressions, which must deal with pointer comparisons without necessarily knowing the runtime layout of the objects involved. A better definition usable at compile time is needed.

Proposed resolution (November, 2014):

Change 8.10 [expr.eq] paragraph 2, converting the existing running text into bullets, as follows:

> If at least one of the operands is a pointer, pointer conversions (7.11 [conv.ptr]) and qualification conversions (7.5 [conv.qual]) are performed on both operands to bring them to their composite pointer type (Clause 8 [expr]). Comparing pointers is defined as follows: ~~Two pointers compare equal~~
>
> - **If one pointer represents the address of a complete object, and another pointer represents the address one past the last element of a different complete object [*Footnote:* An object that is not an array element is considered to belong**

to a single-element array for this purpose; see 8.3.1 [expr.unary.op]. —*end footnote*], the result of the comparison is unspecified.

- **Otherwise,** if ~~they~~ **the pointers** are both null, both point to the same function, or both represent the same address (6.9.2 [basic.compound]), **they compare equal.**
- ~~otherwise they~~ **Otherwise, the pointers** compare unequal.

1858. Comparing pointers to union members

Section: 8.10 [expr.eq] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-02-12

[Moved to DR at the November, 2014 meeting.]

Comparison of pointers to members is currently specified in 8.10 [expr.eq] paragraph 3 as,

> two pointers to members compare equal if they would refer to the same member of the same most derived object (4.5 [intro.object]) or the same subobject if indirection with a hypothetical object of the associated class type were performed, otherwise they compare unequal.

The "same member" requirement could be interpreted as incorrect for union members. The wording should be clarified in this regard.

Proposed Resolution (July, 2014):

Insert the following before bullet 5 of 8.10 [expr.eq] paragraph 3:

- ...
- **If both refer to (possibly different) members of the same union (12.3 [class.union]), they compare equal.**
- Otherwise, two pointers to members compare equal if...

1805. Conversions of array operands in *conditional-expression*s

Section: 8.16 [expr.cond] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2013-11-02

[Moved to DR at the November, 2014 meeting.]

The final bullet of 8.16 [expr.cond] paragraph 3, describing the attempt to convert the operands of the conditional operator to the other operand's type as part of determining the type of the result, says,

- Otherwise (i.e., if `E1` or `E2` has a nonclass type, or if they both have class types but the underlying classes are not either the same or one a base class of the other): `E1` can be converted to match `E2` if `E1` can be implicitly converted to the type that expression `E2` would have if `E2` were converted to a prvalue (or the type it has, if `E2` is a prvalue).

The phrase "if `E2` were converted to a prvalue" is problematic if `E2` has an array type. For example,

```
struct S {
  S(const char *s);
  operator const char *();
};

S s;
const char *f(bool b) {
  return b ? s : "";   // #1
}
```

One might expect that the expression in #1 would be ambiguous, since `S` can be converted both to and from `const char*`. However, the target type for the conversion of `s` is `const char[1]`, not `const char*`, so that conversion fails and the result of the *conditional-expression* has type `S`.

It might be better to specify the target type for this trial conversion to be the type after the usual lvalue-to-rvalue, array-to-pointer, and function-to-pointer conversions instead of simply the result of converting "to a prvalue."

Proposed resolution (February, 2014):

Change the final subbullet of 8.16 [expr.cond] paragraph 3 as follows:

> ...The process for determining whether an operand expression `E1` of type `T1` can be converted to match an operand expression `E2` of type `T2` is defined as follows:
>
> - ...
> - If `E2` is a prvalue or if neither of the conversions above can be done and at least one of the operands has (possibly cv-qualified) class type:

- if `E1` and `E2` have class type...
- Otherwise (i.e., if `E1` or `E2` has a nonclass type, or if they both have class types but **neither are** the underlying classes ~~are not either~~ the same ~~or~~ **nor is** one a base class of the other): `E1` can be converted to match `E2` if `E1` can be implicitly converted to the type that ~~expression~~ `E2` would have ~~if `E2` were converted to a prvalue (or the type it has, if `E2` is a prvalue)~~ **after applying the lvalue-to-rvalue (7.1 [conv.lval]), array-to-pointer (7.2 [conv.array]), and function-to-pointer (7.3 [conv.func]) standard conversions**.

[Editorial note: this wording was approved by CWG, but I'd suggest an editorial change to "...or if both have class types but the underlying classes are not the same and neither is a base class of the other." —wmm]

1843. Bit-field in conditional operator with `throw` operand

Section: 8.16 [expr.cond] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-01-25

[Moved to DR at the November, 2014 meeting.]

Presumably the result of something like

```
b ? x : throw y
```

is a bit-field if `x` is, but the current wording does not say that.

Proposed resolution (February, 2014):

Change 8.16 [expr.cond] paragraph 2 as follows (this assumes the revised wording of the resolution of issue 1299 as the base text):

> If either the second or the third operand has type `void`, one of the following shall hold:
>
> - The second or the third operand (but not both) is a (possibly parenthesized) *throw-expression* (18.1 [except.throw]); the result is of the type and value category of the other operand. The *conditional-expression* is a temporary expression if that operand is a temporary expression **and is a bit-field if that operand is a bit-field**.
> - ...

1895. Deleted conversions in conditional operator operands

Section: 8.16 [expr.cond] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-03-17

[Adopted at the February, 2016 meeting.]

In an example like,

```
struct B;
struct A { A(); A(B&) = delete; operator B&(); };
struct B : A {} b;
B &c = true ? A() : b;
```

the rules of 8.16 [expr.cond] paragraph 3 make this ambiguous: `A()` can be implicitly converted to the type "lvalue reference to `B`," and `b` satisfies the constraints to be converted to an `A` prvalue (it's of a type derived from `A` and the cv-qualifiers are okay). Bullet 3 bullet 1 is clear that we do not actually try to create an `A` temporary from `b`, so we don't notice that it invokes a deleted constructor and rule out that conversion.

If the deleted conversion is in the other sense, the result is unambiguous:

```
struct B;
struct A { A(); A(B&); operator B&() = delete; };
struct B : A {} b;
B &c = true ? A() : b;
```

`A()` can no longer be implicitly converted to the type "lvalue reference to `B`": since the declaration `B &t = A();` is not well formed (it invokes a deleted function), there is no implicit conversion. So we unambiguously convert the third operand to an `A` prvalue.

These should presumably either both be valid or both invalid. EDG and gcc call both ambiguous.

Notes from the June, 2014 meeting:

The wording should be changed to handle the convertibility test more like overload resolution: the conversion "exists" if the conversion function is declared, but is ill-formed if it would actually be used.

Proposed resolution (October, 2015):

1. Add the following as a new paragraph following 8.16 [expr.cond] paragraph 2:

> **Otherwise, if the second and third operand are glvalue bit-fields of the same value category and of types *cv1* `T` and *cv2* `T`, respectively, the operands are considered to be of type *cv* `T` for the remainder of this section, where *cv* is the union of *cv1* and *cv2*.**

2. Change 8.16 [expr.cond] paragraph 3 as follows:

> Otherwise, if the second and third operand have different types and either has (possibly cv-qualified) class type, or if both are glvalues of the same value category and the same type except for cv-qualification, an attempt is made to ~~convert~~ **form an implicit conversion sequence (16.3.3.1 [over.best.ics]) from** each of those operands to the type of the other. **[*Note:* Properties such as access, whether an operand is a bit-field, or whether a conversion function is deleted are ignored for that determination. —*end note*]** ~~The process for determining whether an operand expression E1 of type T1 can be converted to match an operand expression E2 of type T2 is defined as follows:~~ **Attempts are made to form an implicit conversion sequence from an operand expression E1 of type T1 to a target type related to the type T2 of the operand expression E2 as follows:**
>
> - If E2 is an lvalue~~: E1 can be converted to match E2 if E1 can be implicitly converted (Clause 7 [conv]) to the type~~**, the target type is** "lvalue reference to T2", subject to the constraint that in the conversion the reference must bind directly (11.6.3 [dcl.init.ref]) to an lvalue.
> - If E2 is an xvalue~~: E1 can be converted to match E2 if E1 can be implicitly converted to the type~~**, the target type is** "rvalue reference to T2", subject to the constraint that the reference must bind directly.
> - If E2 is a prvalue or if neither of the ~~conversions~~ **conversion sequences** above can be ~~done~~ **formed** and at least one of the operands has (possibly cv-qualified) class type:
>
>   - ~~if E1 and E2 have class type, and the underlying class types are the same or one is a base class of the other: E1 can be converted to match E2 if the class of T2 is the same type as, or a base class of, the class of T1, and the cv-qualification of T2 is the same cv-qualification as, or a greater cv-qualification than, the cv-qualification of T1. If the conversion is applied, E1 is changed to a prvalue of type T2 by copy-initializing a temporary of type T2 from E1 and using that temporary as the converted operand.~~ **if T1 and T2 are the same class type (ignoring cv-qualification), or one is a base class of the other, and T2 is at least as cv-qualified as T1, the target type is T2,**
>   - ~~Otherwise (if E1 or E2 has a non-class type, or if they both have class types but the underlying classes are not the same and neither is a base class of the other): E1 can be converted to match E2 if E1 can be implicitly converted to~~ **otherwise, the target type is** the type that E2 would have after applying the lvalue-to-rvalue (7.1 [conv.lval]), array-to-pointer (7.2 [conv.array]), and function-to-pointer (7.3 [conv.func]) standard conversions.
>
> Using this process, it is determined whether **an implicit conversion sequence can be formed from** the second operand ~~can be converted to match~~ **to the target type determined for** the third operand, and ~~whether the third operand can be converted to match the second operand~~ **vice versa**. If both ~~can be converted~~ **sequences can be formed**, or one ~~can be converted but the conversion is~~ **formed, but it is the** ambiguous **conversion sequence**, the program is ill-formed. If ~~neither can be converted~~ **no conversion sequence can be formed**, the operands are left unchanged and further checking is performed as described below. ~~If exactly one conversion is possible,~~ **Otherwise, if exactly one conversion sequence can be formed,** that conversion is applied to the chosen operand and the converted operand is used in place of the original operand for the remainder of this section. **[*Note:* The conversion might be ill-formed even if an implicit conversion sequence could be formed. —*end note*]**

This resolution also resolves issue 1932.

1932. Bit-field results of conditional operators

Section: 8.16 [expr.cond] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-02-21

[Adopted at the February, 2016 meeting.]

According to 8.16 [expr.cond] paragraph 3,

> if the second and third operand have different types and either has (possibly cv-qualified) class type, or if both are glvalues of the same value category and the same type except for cv-qualification, an attempt is made to convert each of those operands to the type of the other. The process for determining whether an operand expression E1 of type T1 can be converted to match an operand expression E2 of type T2 is defined as follows:
>
> - If E2 is an lvalue: E1 can be converted to match E2 if E1 can be implicitly converted (Clause 7 [conv]) to the type "lvalue reference to T2", subject to the constraint that in the conversion the reference must bind directly (11.6.3 [dcl.init.ref]) to an lvalue.

If two bit-field glvalues have exactly the same scalar type, paragraph 3 does not apply (two non-class operands must differ in at least cv-qualification). For an example like

```
struct S {
  int i:3;
  const int j:4;
} s;
int k = true ? s.i : s.j;
```

the condition is satisfied. The intent is that `S::i` can be converted to `const int` but `S::j` cannot be converted to `int`, so the result should be a bit-field lvalue of type `const int`. However, the test for convertibility is phrased in terms of direct reference binding, which is inapplicable to bit-fields, resulting in neither conversion succeeding, leading to categorizing the expression as ambiguous.

Proposed resolution (October, 2015):

This issue is resolved by the resolution of issue 1895.

1925. Bit-field prvalues

Section: 8.19 [expr.comma] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-05-12

[Moved to DR at the May, 2015 meeting.]

According to 8.19 [expr.comma] paragraph 1,

> The type and value of the result are the type and value of the right operand; the result is of the same value category as its right operand, and is a bit-field if its right operand is a glvalue and a bit-field.

The description of a bit-field result seems to indicate that the operand might not be a glvalue but could still be a bit-field. There doesn't appear to be a normative prohibition against prvalue bit-fields, so one should presumably be added, and this wording should be adjusted to remove the suggestion that such a thing might exist.

Proposed resolution (November, 2014):

1. Change 6.10 [basic.lval] paragraph 2 as follows:

> Whenever a glvalue appears in a context where a prvalue is expected, the glvalue is converted to a prvalue; see 7.1 [conv.lval], 7.2 [conv.array], and 7.3 [conv.func]. [*Note:* An attempt to bind an rvalue reference to an lvalue is not such a context; see 11.6.3 [dcl.init.ref]. —*end note*] **[*Note:* There are no prvalue bit-fields; if a bit-field is converted to a prvalue (7.1 [conv.lval]), a prvalue of the type of the bit-field is created, which might then be promoted (7.6 [conv.prom]). —*end note*]**

2. Change 8.19 [expr.comma] paragraph 1 as follows:

> ...The type and value of the result are the type and value of the right operand; the result is of the same value category as its right operand, and is a bit-field if its right operand is ~~a glvalue and~~ a bit-field. If the value...

1683. Incorrect example after `constexpr` changes

Section: 8.20 [expr.const] **Status:** CD4 **Submitter:** Daniel Krügler **Date:** 2013-05-15

The example in 8.20 [expr.const] paragraph 6,

```
struct A {
  constexpr A(int i) : val(i) { }
  constexpr operator int() { return val; }
  constexpr operator long() { return 43; }
private:
  int val;
};
template<int> struct X { };
constexpr A a = 42;
X<a> x;               // OK: unique conversion to int
int ary[a];           // error: ambiguous conversion
```

is no longer correct now that `constexpr` does not imply `const` for member functions, since the conversion functions cannot be invoked for the constant `a`.

Notes from the September, 2013 meeting:

This issue is being handled editorially and is being placed in "review" status to ensure that the change has been made.

1694. Restriction on reference to temporary as a constant expression

Section: 8.20 [expr.const] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2013-05-30

[Moved to DR at the November, 2014 meeting.]

We're missing a restriction on the value of a temporary which is bound to a static storage duration reference:

```
void f(int n) {
  static constexpr int *&&r = &n;
}
```

This is currently valid, because `&n` is a core constant expression, and it is a constant expression because the reference binds to a temporary (of type `int*`) that has static storage duration (because it's lifetime-extended by the reference binding).

The value of `r` *is* constant here (it's a constant reference to a temporary with a non-constant initializer), but I don't think we should accept this. Generally, I think a temporary which is lifetime-extended by a constexpr variable should also be treated as if it were declared to be a constexpr object.

Proposed resolution (September, 2013) [SUPERSEDED]:

Change 8.20 [expr.const] paragraph 4 as follows:

> A *constant expression* is either a glvalue core constant expression whose value refers to an ~~object with static storage duration or to a function~~ **entity that is a permitted result of a constant expression**, or a prvalue core constant expression whose value is an object where, for that object and its subobjects:
>
> - each non-static data member of reference type refers to an ~~object with static storage duration or to a function~~ **entity that is a permitted result of a constant expression**, and
> - if the object or subobject is of pointer type, it contains the address of an object with static storage duration, the address past the end of such an object (8.7 [expr.add]), the address of a function, or a null pointer value.
>
> **An entity is a permitted result of a constant expression if it is an object with static storage duration that is either not a temporary or is a temporary whose value satisfies the above constraints, or it is a function.**

Proposed resolution (February, 2014):

Change 8.20 [expr.const] paragraph 4 as follows:

> A *constant expression* is either a glvalue core constant expression whose value refers to an ~~object with static storage duration or to a function~~ **entity that is a permitted result of a constant expression (as defined below)**, or a prvalue core constant expression whose value is an object where, for that object and its subobjects:
>
> - each non-static data member of reference type refers to an ~~object with static storage duration or to a function~~ **entity that is a permitted result of a constant expression**, and
> - if the object or subobject is of pointer type, it contains the address of an object with static storage duration, the address past the end of such an object (5.7), the address of a function, or a null pointer value.
>
> **An entity is a *permitted result of a constant expression* if it is an object with static storage duration that is either not a temporary object or is a temporary object whose value satisfies the above constraints, or it is a function.**

1757. Const integral subobjects

Section: 8.20 [expr.const] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2013-09-20

[Moved to DR at the November, 2014 meeting.]

The requirements for a constant expression in 8.20 [expr.const] permit an lvalue-to-rvalue conversion on

> a non-volatile glvalue of integral or enumeration type that refers to a non-volatile const object with a preceding initialization, initialized with a constant expression

This does not exclude subobjects of objects that are not compile-time constants, for example:

```
int f();
struct S {
  S() : a(f()), b(5) {}
  int a, b;
};
const S s;
constexpr int k = s.b;
```

This rule is intended to provide backward compatibility with pre-`constexpr` C++, but it should be restricted to complete objects. Care should be taken in resolving this issue not to break the handling of string literals, since use of their elements in constant expressions depends on the current form of this rule.

Proposed resolution (February, 2014):

Change 8.20 [expr.const] paragraph 2 bullet 7 as follows:

> A *conditional-expression* `e` is a *core constant expression* unless the evaluation of `e`, following the rules of the abstract machine (4.6 [intro.execution]), would evaluate one of the following expressions:
>
> - ...
> - an lvalue-to-rvalue conversion (7.1 [conv.lval]) unless it is applied to
>   - a non-volatile glvalue of integral or enumeration type that refers to a **complete** non-volatile const object with a preceding initialization, initialized with a constant expression ~~[*Note:* a string literal (5.13.5 [lex.string]) corresponds to an array of such objects. — *end note*]~~, or
>   - **a non-volatile glvalue that refers to a subobject of a string literal (5.13.5 [lex.string]), or**

- a non-volatile glvalue that refers to a non-volatile object defined with `constexpr`...

1952. Constant expressions and library undefined behavior

Section: 8.20 [expr.const] **Status:** CD4 **Submitter:** Daniel Krügler **Date:** 2014-06-22

[Moved to DR at the May, 2015 meeting.]

According to bullet 2.5 of 8.20 [expr.const],

> A *conditional-expression* `e` is a core constant expression unless the evaluation of `e`, following the rules of the abstract machine (4.6 [intro.execution]), would evaluate one of the following expressions:
>
> - ...
> - an operation that would have undefined behavior [*Note:* including, for example, signed integer overflow (Clause 8 [expr]), certain pointer arithmetic (8.7 [expr.add]), division by zero (8.6 [expr.mul]), or certain shift operations (8.8 [expr.shift]) — *end note*];
> - ...

The definition of "operation" is unclear. In particular, is it intended to include use of library components that are specified to produce undefined behavior, such as use of the `offsetof` macro when applied to a non-standard-layout class?

Proposed resolution (April, 2015):

1. Change 8.20 [expr.const] bullet 2.5 as follows:
   - an operation that would have undefined behavior **as specified in Clauses 4 [intro] through 19 [cpp] of this International Standard** [*Note:* including, for example, signed integer overflow (Clause 8 [expr]), certain pointer arithmetic (8.7 [expr.add]), division by zero (8.6 [expr.mul]), or certain shift operations (8.8 [expr.shift]) — *end note*];
2. Add the following at the end of 8.20 [expr.const] paragraph 2:

   If `e` satisfies the constraints of a core constant expression, but evaluation of `e` would evaluate an operation that has undefined behavior as specified in Clauses 20 [library] through 33 [thread] of this International Standard, it is unspecified whether `e` is a core constant expression.

2004. Unions with mutable members in constant expressions

Section: 8.20 [expr.const] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-09-16

[Moved to DR at the October, 2015 meeting.]

In an example like

```
union U { int a; mutable int b; };
constexpr U u1 = {1};
int k = (u1.b = 2);
constexpr U u2 = u1; // ok!!
```

The initialization of `u2` is not disqualified by the current wording of the Standard because the copy is done via the object representation, not formally involving an lvalue-to-rvalue conversion. A restriction should be added to 8.20 [expr.const] forbidding the evaluation of a defaulted copy/move construction/assignment on a class type that has any variant mutable subobjects.

Proposed resolution (May, 2015):

1. Add the following bullet after bullet 3.1 of 10.1.5 [dcl.constexpr]:

   The definition of a `constexpr` function shall satisfy the following constraints:
   - it shall not be virtual (13.3 [class.virtual]);
   - **for a defaulted copy/move assignment, the class of which it is a member shall not have a mutable subobject that is a variant member;**
   - ...
2. Add the following bullet after bullet 4.1 of 10.1.5 [dcl.constexpr]

   The definition of a `constexpr` constructor shall satisfy the following constraints:
   - the class shall not have any virtual base classes;
   - **for a defaulted copy/move constructor, the class shall not have a mutable subobject that is a variant member;**

- ...

2022. Copy elision in constant expressions

Section: 8.20 [expr.const] **Status:** CD4 **Submitter:** Jason Merrill **Date:** 2014-10-16

[Adopted at the June, 2016 meeting.]

Consider the following example:

```
struct A {
  void *p;
  constexpr A(): p(this) {}
};

constexpr A a;               // well-formed
constexpr A b = A();         // ?
```

The declaration of `a` seems well-formed because the address of `a` is constant.

The declaration of `b`, however, seems to depend on whether copy elision is performed. If it is, the declaration is the equivalent of `a`; if not, however, this creates a temporary and initializes `p` to the address of that temporary, making the initialization non-constant and the declaration ill-formed.

It does not seem desirable for the well-formedness of the program to depend on whether the implementation performs an optional copy elision.

Notes from the November, 2014 meeting:

CWG decided to leave it unspecified whether copy elision is performed in cases like this and to add a note to 15.8 [class.copy] to make clear that that outcome is intentional.

Notes from the May, 2015 meeting:

CWG agreed that copy elision should be mandatory in constant expressions.

Proposed resolution (April, 2016):

1. Change 8.20 [expr.const] paragraph 1 as follows:

   ...Expressions that satisfy these requirements**, assuming that copy elision is performed,** are called *constant expressions*. [*Note:*...

2. Change 10.1.5 [dcl.constexpr] paragraph 7 as follows, breaking the existing running text into a bulleted list:

   A call to a `constexpr` function produces the same result as a call to an equivalent non-`constexpr` function in all respects except that

   - a call to a `constexpr` function can appear in a constant expression **(8.20 [expr.const]) and**
   - **copy elision is mandatory in a constant expression (15.8 [class.copy])**.

3. Change 15.8 [class.copy] paragraph 31 as follows:

   ...This elision of copy/move operations, called copy elision, is permitted in the following circumstances (which may be combined to eliminate multiple copies):

   - ...

   Copy elision is required where an expression is evaluated in a context requiring a constant expression (8.20 [expr.const]) and in constant initialization (6.6.2 [basic.start.static]). [*Note:* Copy elision might not be performed if the same expression is evaluated in another context. —*end note*] [*Example:*

   ```
   class Thing {
   public:
     Thing();
     ~Thing();
     Thing(const Thing&);
   };

   Thing f() {
     Thing t;
     return t;
   }

   Thing t2 = f();

   struct A {
     void *p;
     constexpr A(): p(this) {}
   };

   constexpr A a;         // well-formed, a.p points to a
   constexpr A b = A();   // well-formed, b.p points to b
   ```

```
void g() {
  A c = A();            // well-formed, c.p may point to c or to an ephemeral temporary
}
```

Here the criteria for elision can be combined...

2129. Non-object prvalues and constant expressions

Section: 8.20 [expr.const] **Status:** CD4 **Submitter:** Faisal Vali **Date:** 2015-05-20

[Adopted at the February, 2016 meeting.]

According to 8.20 [expr.const] paragraph 5,

> A constant expression is either a glvalue core constant expression whose value... or a prvalue core constant expression whose value is an object where...

Since an integer literal is prvalue that is not an object, this definition does not allow it to be a constant expression.

Proposed resolution (February, 2016):

Change 8.20 [expr.const] paragraph 5 as follows:

> A *constant expression* is either a glvalue core constant expression whose value refers to an entity that is a permitted result of a constant expression (as defined below), or a prvalue core constant expression whose value ~~is an object where, for that object and its subobjects~~ **satisfies the following constraints**:
>
> - **if the value is an object of class type,** each non-static data member of reference type refers to an entity that is a permitted result of a constant expression, ~~and~~
> - if the ~~object or subobject~~ **value** is of pointer type, it contains the address of an object with static storage duration, the address past the end of such an object (8.7 [expr.add]), the address of a function, or a null pointer value~~.~~**, and**
> - **if the value is an object of class or array type, each subobject satisfies these constraints for the value.**
>
> An entity is a *permitted result of a constant expression* if...

2167. Non-member references with lifetimes within the current evaluation

Section: 8.20 [expr.const] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2015-08-11

[Adopted at the February, 2016 meeting.]

The current wording of 8.20 [expr.const] bullet 2.9 says:

- an *id-expression* that refers to a variable or data member of reference type unless the reference has a preceding initialization and either
  - it is initialized with a constant expression or
  - it is a non-static data member of an object whose lifetime began within the evaluation of `e`;

This incorrectly excludes non-member references whose lifetime began within the current evaluation.

Proposed resolution (February, 2016):

Change 8.20 [expr.const] bullet 2.9.2 as follows:

> A *conditional-expression* e is a *core constant expression* unless the evaluation of e, following the rules of the abstract machine (4.6 [intro.execution]), would evaluate one of the following expressions:
>
> - ...
> - an *id-expression* that refers to a variable or data member of reference type unless the reference has a preceding initialization and either
>   - it is initialized with a constant expression or
>   - ~~it is a non-static data member of an object whose~~ **its** lifetime began within the evaluation of `e`;
> - ...

1274. Common nonterminal for *expression* and *braced-init-list*

Section: 9.5.4 [stmt.ranged] **Status:** CD4 **Submitter:** Daniel Krügler **Date:** 2011-03-25

[Moved to DR at the October, 2015 meeting.]

It would be helpful to have a single grammar term for *expression* and *braced-init-list*, which often occur together in the text. In particular, 9.5.4 [stmt.ranged] paragraph 1 allows both, but the description of `__RangeT` refers only to the *expression* case; such errors would be less likely if the common term were available.

Proposed resolution (May, 2015):

1. Add a new production to the grammar in 11.6 [dcl.init] paragraph 1:

expr-or-braced-init-list:
expression
braced-init-list

2. Change the grammar in 8.2 [expr.post] paragraph 1 as follows:

postfix-expression:
primary-expression
postfix-expression [~~*expression*~~ ***expr-or-braced-init-list***]
~~*postfix-expression* . *braced-init-list*~~
postfix-expression (*expression-list*$_{opt}$)
...

3. Change the grammar in 9.5 [stmt.iter] paragraph 1 as follows:

for-range-initializer:
~~*expression*~~
~~*braced-init-list*~~
expr-or-braced-init-list

4. Change 9.5.4 [stmt.ranged] paragraph 1 as follows:

~~For a range-based `for` statement of the form~~

~~`for (` *for-range-declaration* `:` *expression* `)` *statement*~~

~~let *range-init* be equivalent to the *expression* surrounded by parentheses~~[90]

~~`(` *expression* `)`~~

~~and for a range-based `for` statement of the form~~

~~`for (` *for-range-declaration* `:` *braced-init-list* `)` *statement*~~

~~let *range-init* be equivalent to the *braced-init-list*. In each case, a~~ **A** range-based for statement is equivalent to

```
{
auto && __range = range-init for-range-initializer;
  for ( auto __begin = begin-expr,
             __end = end-expr;
        __begin != __end;
        ++__begin ) {
    for-range-declaration = *__begin;
    statement
  }
}
```

where

- **if the *for-range-initializer* is an *expression*, it is regarded as if it were surrounded by parentheses (so that a comma operator cannot be reinterpreted as delimiting two *init-declarator*s);**
- `__range`, `__begin`, and `__end` are variables defined for exposition only; and
- ~~`_RangeT` is the type of the expression, and~~
- *begin-expr* and *end-expr* are determined as follows:
  - if ~~`_RangeT`~~ **the *for-range-initializer*** is an **expression of** array type `R`, *begin-expr* and *end-expr* are `__range` and `__range + __bound`, respectively, where `__bound` is the array bound. If ~~`_RangeT`~~ `R` is an array of unknown size or an array of incomplete type, the program is ill-formed;
  - if ~~`_RangeT`~~ **the *for-range-initializer*** is ~~a~~ **an expression of** class type `C`, the *unqualified-id*s `begin` and `end` are looked up in the scope of ~~class `_RangeT`~~ `C` as if by class member access lookup (6.4.5 [basic.lookup.classref]), and if either (or both) finds at least one declaration, *begin-expr* and *end-expr* are `__range.begin()` and `__range.end()`, respectively;
  - otherwise, *begin-expr* and *end-expr* are `begin(__range)` and `end(__range)`, respectively, where `begin` and `end` are looked up in the associated namespaces (6.4.2 [basic.lookup.argdep]). [*Note:* Ordinary unqualified lookup (6.4.1

[basic.lookup.unqual]) is not performed. —*end note*]

5. Change the grammar of 9.6 [stmt.jump] paragraph 1 as follows:

jump-statement:
`break` ;
`continue` ;
`return` ~~*expression*~~ ***expr-or-braced-init-list***$_{opt}$;
~~`return` *braced-init-list* ;~~
`goto` *identifier* ;

6. Change 9.6.3 [stmt.return] paragraph 2 as follows:

The ~~*expression* or *braced-init-list*~~ ***expr-or-braced-init-list*** of a `return` statement is called its operand. A return statement...

7. Change 16.5.5 [over.sub] paragraph 1 as follows:

`operator[]` shall be a non-static member function with exactly one parameter. It implements the subscripting syntax

postfix-expression `[` ~~*expression*~~ ***expr-or-braced-init-list*** `]`

~~or~~

~~*postfix-expression* `[` *braced-init-list* `]`~~

Thus, a subscripting expression...

2017. Flowing off end is not equivalent to no-expression return

Section: 9.6.3 [stmt.return] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-10-06

[Adopted at the February, 2016 meeting.]

According to 9.6.3 [stmt.return] paragraph 2,

> Flowing off the end of a function is equivalent to a `return` with no value...

This is not correct, since a `return` with no value is ill-formed in a value-returning function but flowing off the end results in undefined behavior.

Proposed resolution (May, 2015): [SUPERSEDED]

Change 9.6.3 [stmt.return] paragraph 2 as follows:

> ...Flowing off the end of a **value-returning** function is **undefined behavior. Flowing off the end of any other function is** equivalent to a return with no value~~; this results in undefined behavior in a value-returning function~~.

Additional notes, October, 2015:

There is similar wording in 18.3 [except.handle] paragraph 14. Also, it might be better to avoid the use of the word "value", since it is currently not clearly defined.

Proposed resolution (October, 2015):

1. Change 6.6.1 [basic.start.main] paragraph 5 as follows:

> A return statement in `main` has the effect of leaving the main function (destroying any objects with automatic storage duration) and calling `std::exit` with the return value as the argument. If control ~~reaches~~ **flows off** the end of **the *compound-statement* of** `main` ~~without encountering a `return` statement~~, the effect is ~~that of executing~~ **equivalent to a** `return` **with operand** `0` **(see also 18.3 [except.handle]).**
>
> ~~`return 0;`~~

2. Change 9.6.3 [stmt.return] paragraph 2 as follows:

> ...Flowing off the end of a function **with a void return type** is equivalent to a `return` with no ~~value; this results in undefined behavior in a value-returning function~~ **operand**. **Otherwise, flowing off the end of a function other than** `main` **(6.6.1 [basic.start.main] results in undefined behavior.**

3. Change 18.3 [except.handle] paragraph 14 as follows:

> The currently handled exception is rethrown if control reaches the end of a handler of the *function-try-block* of a constructor or destructor. Otherwise, ~~a function returns when control reaches the end of a handler for the *function-try-block* (9.6.3 [stmt.return]). Flowing off the end of a *function-try-block* is equivalent to a `return` with no value; this results in undefined behavior in a value-returning function (9.6.3 [stmt.return])~~ **flowing off the end of the *compound-statement* of a *handler* of a *function-try-block* is equivalent to flowing off the end of the *compound-statement* of that function (see 9.6.3 [stmt.return])**.

1830. Repeated specifiers

Section: 10 [dcl.dcl] **Status:** CD4 **Submitter:** Ville Voutilainen **Date:** 2014-01-10

[Moved to DR at the November, 2014 meeting.]

Although repeated *type-specifier*s such as `const` are forbidden, there is no such prohibition against repeated non-type specifiers like `constexpr` and `virtual`. Should there be?

On the "con" side, it's not clear that such a prohibition actually helps anyone; it could happen via macros, and a warning about non-macro use could be a QoI issue. Also, C99 moved in the opposite direction, removing the prohibition against repeated cv-qualifiers.

Proposed resolution (February, 2014):

Add the following as a new paragraph before 10.1 [dcl.spec] paragraph 2:

> **Each *decl-specifier* shall appear at most once in the complete *decl-specifier-seq* of a declaration, except that `long` may appear twice.**
>
> If a *type-name* is encountered...

1990. Ambiguity due to optional *decl-specifier-seq*

Section: 10 [dcl.dcl] **Status:** CD4 **Submitter:** Hubert Tong **Date:** 2014-08-27

[Moved to DR at the October, 2015 meeting.]

In an example like

```
void f() {
  f();  // #1
}
```

The statement at #1 is ambiguous and can be parsed as either an expression or as a declaration. The problem is the fact that the *decl-specifier-seq* in a *simple-declaration* is optional.

Proposed resolution (May, 2015):

1. Change the grammar in 10 [dcl.dcl] paragraph 1 as follows:

> *declaration:*
> *block-declaration*
> ***nodeclspec-function-declaration***
> *function_definition*
> ...
>
> ***nodeclspec-function-declaration:***
> ***attribute-specifier-seq*$_{opt}$ *declarator* ;
>
> *alias-declaration:*
> `using` *identifier attribute-specifier-seq*$_{opt}$ `=` *type-id* ;
>
> *simple-declaration:*
> *decl-specifier-seq*$_{opt}$ *init-declarator-list* ;
> *attribute-specifier-seq decl-specifier-seq*$_{opt}$ *init-declarator-list* ;
> ...

2. Change 10 [dcl.dcl] paragraph 2 as follows:

> ~~The~~ **A** *simple-declaration* **or *nodeclspec-function-declaration* of the form**
>
> *attribute-specifier-seq*$_{opt}$ *decl-specifier-seq*$_{opt}$ *init-declarator-list*$_{opt}$;
>
> is divided into three parts. Attributes are described in 10.6 [dcl.attr]. *decl-specifier*s, the principal components of a *decl-specifier-seq*, are described in 10.1 [dcl.spec]. declarators, the components of an *init-declarator-list*, are described in Clause 11 [dcl.decl]. The *attribute-specifier-seq* ~~in a *simple-declaration*~~ appertains to each of the entities declared by the declarators of the *init-declarator-list*. [*Note:*...

3. Change 10 [dcl.dcl] paragraph 11 as follows:

> ~~Only in function declarations for~~ **A *nodeclspec-function-declaration* shall declare a** constructor~~s~~, destructor~~s~~, ~~and type~~ **or** conversion~~s~~ **function** ~~can the *decl-specifier-seq* be omitted~~.[93] **[*Note:* a *nodeclspec-function-declaration* can only be**

used in a *template-declaration* (Clause 17 [temp]), *explicit-instantiation* (17.8.2 [temp.explicit]), or *explicit-specialization* (17.8.3 [temp.expl.spec]). —*end note*]

4. Change 11.3 [dcl.meaning] paragraph 1 as follows:

> ~~A list of declarators appears after an optional (Clause 10 [dcl.dcl]) *decl-specifier-seq* (10.1 [dcl.spec]). Each~~ **A** declarator contains exactly one *declarator-id*, it names the identifier...

5. Change 15.1 [class.ctor] paragraph 1 as follows:

> ...~~In a constructor declaration, each~~ **Each** *decl-specifier* in the ~~optional~~ *decl-specifier-seq* **of a constructor declaration (if any)** shall be `friend`, `inline`, `explicit`, or `constexpr`. [*Example*:...

6. Change 15.3.2 [class.conv.fct] paragraph 1 as follows:

> ...Such functions are called conversion functions. ~~No return type can be specified.~~ **A *decl-specifier* in the *decl-specifier-seq* of a conversion function (if any) shall be neither a *type-specifier* nor** `static`**.** ~~If a conversion function is a member function, the~~ **The** type of the conversion function (11.3.5 [dcl.fct]) is...

7. Delete 15.3.2 [class.conv.fct] paragraph 6:

> ~~Conversion functions cannot be declared `static`.~~

8. Change 15.4 [class.dtor] paragraph 1 as follows:

> ...~~In a destructor declaration, each~~ **Each** *decl-specifier* of the ~~optional~~ *decl-specifier-seq* **of a destructor declaration (if any)** shall be `friend`, `inline`, or `virtual`.

This resolution also resolves issue 2016.

1793. `thread_local` in explicit specializations

Section: 10.1.1 [dcl.stc] **Status:** CD4 **Submitter:** Mike Herrick **Date:** 2013-10-02

[Moved to DR at the November, 2014 meeting.]

According to 10.1.1 [dcl.stc] paragraph 1,

> ...If `thread_local` appears in any declaration of a variable it shall be present in all declarations of that entity... A *storage-class-specifier* shall not be specified in an explicit specialization (17.8.3 [temp.expl.spec]) or an explicit instantiation (17.8.2 [temp.explicit]) directive.

These two requirements appear to be in conflict when an explicit instantiation or explicit specialization names a `thread_local` variable. For example,

```
template <class T> struct S {
  thread_local static int tlm;
};
template <> int S<int>::tlm = 0;
template <> thread_local int S<float>::tlm = 0;
```

which of the two explicit specializations is correct?

Proposed resolution (February, 2014):

Change 10.1.1 [dcl.stc] paragraph 1 as follows:

> ...A *storage-class-specifier* **other than** `thread_local` shall not be specified in an explicit specialization (17.8.3 [temp.expl.spec]) or an explicit instantiation (17.8.2 [temp.explicit]) directive.

1799. `mutable` and non-explicit const qualification

Section: 10.1.1 [dcl.stc] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2013-10-21

[Moved to DR at the November, 2014 meeting.]

According to 10.1.1 [dcl.stc] paragraph 9,

> The `mutable` specifier can be applied only to names of class data members (12.2 [class.mem]) and cannot be applied to names declared `const` or `static`, and cannot be applied to reference members.

This is similar to issue 1686 in that the restriction appears to apply only to declarations in which the `const` keyword appears directly. It should instead apply to members with const-qualified types, regardless of how the qualification was achieved.

Proposed resolution (January, 2014) [SUPERSEDED]:

Change 10.1.1 [dcl.stc] paragraph 9 as follows:

> The `mutable` specifier can be applied only to names of **non-static** class data members (12.2 [class.mem]) ~~and cannot be applied to names declared `const` or `static`, and cannot be applied to reference members~~ **whose type is neither const-qualified nor a reference type**. [*Example:*...

Proposed resolution (February, 2014):

Change 10.1.1 [dcl.stc] paragraph 9 as follows:

> The `mutable` specifier ~~can be applied~~ **shall appear** only ~~to names~~ **in the declaration** of ~~class~~ **a non-static** data ~~members~~ **member** (12.2 [class.mem]) ~~and cannot be applied to names declared `const` or `static`, and cannot be applied to reference members~~ **whose type is neither const-qualified nor a reference type**. [*Example:*...

1930. *init-declarator-list* vs *member-declarator-list*

Section: 10.1.1 [dcl.stc] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-05-19

[Adopted at the February, 2016 meeting.]

According to 10.1.1 [dcl.stc] paragraph 1,

> If a *storage-class-specifier* appears in a *decl-specifier-seq*, there can be no `typedef` specifier in the same *decl-specifier-seq* and the *init-declarator-list* of the declaration shall not be empty...

This obviously should apply to `mutable` but does not because `mutable` applies to *member-declarator-list*s, not *init-declarator-list*s. Similarly, in 10.1.7.1 [dcl.type.cv] paragraph 1,

> If a *cv-qualifier* appears in a *decl-specifier-seq*, the *init-declarator-list* of the declaration shall not be empty.

this should apply to member declarations as well.

Proposed resolution (October, 2015):

1. Change 10.1.1 [dcl.stc] paragraph 1 as follows:

> ...If a *storage-class-specifier* appears in a *decl-specifier-seq*, there can be no `typedef` specifier in the same *decl-specifier-seq* and the *init-declarator-list* **or *member-declarator-list*** of the declaration shall not be empty (except for an anonymous union declared in a named namespace or in the global namespace, which shall be declared `static` (12.3 [class.union])). The *storage-class-specifier* applies...

2. Change 10.1.7.1 [dcl.type.cv] paragraph 1 as follows:

> ...If a *cv-qualifier* appears in a *decl-specifier-seq*, the *init-declarator-list* **or *member-declarator-list*** of the declaration shall not be empty. [*Note:*...

Additional note, November, 2014:

The preceding resolution, which was advanced to "tentatively ready" status during the review session following the November, 2014 (Urbana) meeting, introduces an apparently unintentional conflict with 12.3 [class.union] paragraph 6 regarding the requirements for anonymous unions in unnamed namespaces and has been returned to "review" status to allow further discussion.

Notes from the October, 2015 meeting:

The proposed resolution was changed to address the preceding concern.

1823. String literal uniqueness in inline functions

Section: 10.1.2 [dcl.fct.spec] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2013-12-17

[Moved to DR at the November, 2014 meeting.]

According to 10.1.2 [dcl.fct.spec] paragraph 4,

> A string literal in the body of an extern inline function is the same object in different translation units.

The Standard does not otherwise specify when string literals are required to be the same object, and this requirement is not widely implemented. Should it be removed?

Proposed resolution (February, 2014):

1. Change 5.13.5 [lex.string] paragraph 1 as follows:

> A ~~string literal~~ ***string-literal*** is a sequence of characters...

2. Change 5.13.5 [lex.string] paragraph 2 as follows:

   A ~~string literal~~ ***string-literal*** that has an `R` in the prefix...

3. Change 5.13.5 [lex.string] paragraph 6 as follows:

   After translation phase 6, a ~~string literal~~ ***string-literal*** that does not begin...

4. Change 5.13.5 [lex.string] paragraph 7 as follows:

   A ~~string literal~~ ***string-literal*** that begins with `u8`...

5. Change 5.13.5 [lex.string] paragraph 10 as follows:

   A ~~string literal~~ ***string-literal*** that begins with `u`, such as `u"asdf"`, is a `char16_t` string literal. A `char16_t` string literal has type "array of *n* `const char16_t`", where *n* is the size of the string as defined below; it ~~has static storage duration and~~ is initialized with the given characters. A single *c-char* may produce more than one `char16_t` character in the form of surrogate pairs.

6. Change 5.13.5 [lex.string] paragraph 11 as follows:

   A ~~string literal~~ ***string-literal*** that begins with `U`, such as `U"asdf"`, is a `char32_t` string literal. A `char32_t` string literal has type "array of *n* `const char32_t`", where *n* is the size of the string as defined below; it ~~has static storage duration and~~ is initialized with the given characters.

7. Change 5.13.5 [lex.string] paragraph 12 as follows:

   A ~~string literal~~ ***string-literal*** that begins with `L`, such as `L"asdf"`, is a wide string literal. A wide string literal has type "array of *n* `const wchar_t`", where *n* is the size of the string as defined below; it ~~has static storage duration and~~ is initialized with the given characters.

8. Delete 5.13.5 [lex.string] paragraph 13:

   ~~Whether all string literals are distinct (that is, are stored in nonoverlapping objects) is implementation-defined. The effect of attempting to modify a string literal is undefined.~~

9. Change 5.13.5 [lex.string] paragraph 14 as follows:

   In translation phase 6 (5.2 [lex.phases]), adjacent ~~string literals~~ ***string-literal*s** are concatenated. If both ~~string literals~~ ***string-literal*s** have the same *encoding-prefix*, the resulting concatenated string literal has that *encoding-prefix*. If one ~~string literal~~ ***string-literal*** has no *encoding-prefix*, it is treated as a ~~string literal~~ ***string-literal*** of the same *encoding-prefix* as the other operand. If a UTF-8 string literal token is adjacent to a wide string literal token, the program is ill-formed. Any other concatenations are conditionally-supported with implementation-defined behavior. [*Note:* This concatenation is an interpretation, not a conversion. Because the interpretation happens in translation phase 6 (after each character from a literal has been translated into a value from the appropriate character set), a ~~string literal~~ ***string-literal*** s initial rawness has no effect on the interpretation or well-formedness of the concatenation. —*end note*] Table 8...

10. Add the following as a new paragraph at the end of 5.13.5 [lex.string]:

    Evaluating a *string-literal* results in a string literal object with static storage duration, initialized from the given characters as specified above. Whether all string literals are distinct (that is, are stored in nonoverlapping objects) and whether successive evaluations of a *string-literal* yield the same or a different object is unspecified. [*Note:* The effect of attempting to modify a string literal is undefined. —*end note*]

11. Change 10.1.2 [dcl.fct.spec] paragraph 4 as follows:

    ...A `static` local variable in an `extern inline` function always refers to the same object. ~~A string literal in the body of an `extern inline` function is the same object in different translation units. [*Note:* A string literal appearing in a default argument is not in the body of an inline function merely because the expression is used in a function call from that inline function. —*end note*]~~ A type defined within the body of an `extern inline` function is the same type in every translation unit.

Additional note, February, 2014:

Two editorial changes have been made since CWG approved the proposed resolution:

1. The proposed change to 5.13.5 [lex.string] paragraph 15 has not been made. The term *string-literal* refers to the syntactic construct appearing in the source, but the addition of the terminating null character is made to the concatenated string literal, which is (appropriately) referred to in the preceding paragraph (as in the original text of paragraph 15) using the English term, not the non-terminal.

2. The deletion of the requirement in 10.1.2 [dcl.fct.spec] paragraph 4 that string literals in inline functions be the same made the note following that requirement irrelevant, so the deletion has been extended to include the note as well.

The issue has been returned to "review" status to allow possible reconsideration of these editorial changes.

1247. Restriction on alias name appearing in *type-id*

Section: 10.1.3 [dcl.typedef] **Status:** CD4 **Submitter:** James Widman **Date:** 2011-02-26

[Moved to DR at the May, 2015 meeting.]

With the resolution of issue 1044, there is no need to say that the name of the alias cannot appear in the *type-id* of the declaration.

Proposed resolution (April, 2015):

Change 10.1.3 [dcl.typedef] paragraph 2 as follows:

> A *typedef-name* can also be introduced by an *alias-declaration*. The *identifier* following the `using` keyword becomes a *typedef-name* and the optional *attribute-specifier-seq* following the *identifier* appertains to that *typedef-name*. It has the same semantics as if it were introduced by the `typedef` specifier. In particular, it does not define a new type ~~and it shall not appear in the *type-id*~~. [*Example:*...

2071. `typedef` with no declarator

Section: 10.1.3 [dcl.typedef] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-01-16

[Adopted at the February, 2016 meeting.]

There should be a rule to prohibit the almost certainly erroneous declaration

```
typedef struct X { };    // Missing declarator
```

Proposed resolution (September, 2015):

Change 10.1.3 [dcl.typedef] paragraph 1 as follows:

> Declarations containing the *decl-specifier* `typedef` declare identifiers that can be used later for naming fundamental (6.9.1 [basic.fundamental]) or compound (6.9.2 [basic.compound]) types. The `typedef` specifier shall not be combined in a *decl-specifier-seq* with any other kind of specifier except a *type-specifier*, and it shall not be used in the *decl-specifier-seq* of a *parameter-declaration* (11.3.5 [dcl.fct]) nor in the *decl-specifier-seq* of a *function-definition* (11.4 [dcl.fct.def]). **If a** `typedef` **specifier appears in a declaration without a *declarator*, the program is ill-formed.**

1712. `constexpr` variable template declarations

Section: 10.1.5 [dcl.constexpr] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2013-07-08

[Moved to DR at the November, 2014 meeting.]

According to 10.1.5 [dcl.constexpr] paragraph 1,

> If any declaration of a function, function template, or variable template has a `constexpr` specifier, then all its declarations shall contain the `constexpr` specifier.

This requirement does not make sense applied to variable templates. The `constexpr` specifier requires that there be an initializer, and a variable template declaration with an initializer is a definition, so there cannot be more than one declaration of a variable template with the `constexpr` specifier.

Proposed resolution (February, 2014):

Change 10.1.5 [dcl.constexpr] paragraph 1 as follows:

> ...If any declaration of a function~~,~~ **or** function template~~, or variable template~~ has a `constexpr` specifier, then all its declarations shall contain the `constexpr` specifier. [*Note:*...

1872. Instantiations of `constexpr` templates that cannot appear in constant expressions

Section: 10.1.5 [dcl.constexpr] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-02-17

[Adopted at the February, 2016 meeting.]

According to 10.1.5 [dcl.constexpr] paragraph 6,

> If the instantiated template specialization of a `constexpr` function template or member function of a class template would fail to satisfy the requirements for a `constexpr` function or `constexpr` constructor, that specialization is still a `constexpr` function or `constexpr` constructor, even though a call to such a function cannot appear in a constant expression.

The restriction on appearing in a constant expression assumes the previous wording that made such a specialization non-`constexpr`, and a call to a non-`constexpr` function cannot appear in a constant expression. With the current wording, however, there is no normative restriction against calls to such specializations. 8.20 [expr.const] should be updated to include such a prohibition.

Proposed resolution (January, 2016):

Add the following bullet following 8.20 [expr.const] bulllet 2.3:

> A *conditional-expression* e is a *core constant expression* unless the evaluation of `e`, following the rules of the abstract machine (4.6 [intro.execution]), would evaluate one of the following expressions:
>
> - ...
> - an invocation of an undefined `constexpr` function or an undefined `constexpr` constructor;
> - **an invocation of an instantiated `constexpr` function or `constexpr` constructor that fails to satisfy the requirements for a `constexpr` function or `constexpr` constructor (10.1.5 [dcl.constexpr]);**
> - ...

1911. `constexpr` constructor with non-literal base class

Section: 10.1.5 [dcl.constexpr] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-04-13

[Moved to DR at the November, 2014 meeting.]

An example like

```
struct X {
  std::unique_ptr<int> p;
  constexpr X() { }
};
```

is ill-formed because the `X` constructor cannot be used in a constant expression, because a constant expression cannot construct an object of a non-literal type like `unique_ptr`. This prevents use of something like

```
X x;
```

to guarantee constant-initialization.

Proposed resolution (June, 2014):

Change 10.1.5 [dcl.constexpr] paragraph 5 as follows:

> For a non-template, non-defaulted `constexpr` function or a non-template, non-defaulted, non-inheriting `constexpr` constructor, if no argument values exist such that an invocation of the function or constructor could be an evaluated subexpression of a core constant expression (8.20 [expr.const]), **or, for a constructor, a constant initializer for some object (6.6.2 [basic.start.static]),** the program is ill-formed; no diagnostic required.

2163. Labels in `constexpr` functions

Section: 10.1.5 [dcl.constexpr] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2015-07-24

[Adopted at the February, 2016 meeting.]

The requirements for `constexpr` functions do not, but presumably should, forbid the appearance of a label in the function body (`goto`s are prohibited).

Proposed resolution (January, 2016):

Add the following as an additional bullet following 10.1.5 [dcl.constexpr] bullet 3.5.2:

> The definition of a `constexpr` function shall satisfy the following requirements:
>
> - ...
> - its *function-body* shall be `= delete`, `= default`, or a *compound-statement* that does not contain
>   - an *asm-definition*,
>   - a `goto` statement,
>   - **an identifier label (9.1 [stmt.label]),**
>   - ...

609. What is a "top-level" cv-qualifier?

Section: 10.1.7.1 [dcl.type.cv] **Status:** CD4 **Submitter:** Dawn Perchik **Date:** 5 November 2006

[Moved to DR at the November, 2014 meeting.]

The phrase "top-level cv-qualifier" is used numerous times in the Standard, but it is not defined. The phrase could be misunderstood to indicate that the `const` in something like `const T&` is at the "top level," because where it appears is the highest level at which it is permitted: `T& const` is ill-formed.

Proposed resolution (February, 2014):

Change 6.9.3 [basic.type.qualifier] paragraph 5 as follows, splitting it into two paragraphs:

> In this International Standard, the notation *cv* (or *cv1*, *cv2*, etc.), used in the description of types, represents an arbitrary set of cv-qualifiers, i.e., one of `{const}`, `{volatile}`, `{const, volatile}`, or the empty set. **For a type *cv* `T`, the *top-level cv-qualifiers* of that type are those denoted by *cv*. [*Example:* The type corresponding to the *type-id* "`const int&`" has no top-level cv-qualifiers. The type corresponding to the *type-id* "`volatile int * const`" has the top-level cv-qualifier `const`. For a class type `C`, the type corresponding to the *type-id* "`void (C::* volatile)(int) const`" has the top-level cv-qualifier `volatile`. —*end example*]**
>
> Cv-qualifiers applied to an array type attach...

1600. Erroneous reference initialization in example

Section: 10.1.7.2 [dcl.type.simple] **Status:** CD4 **Submitter:** Niels Dekker **Date:** 2012-12-30

[Moved to DR at the November, 2014 meeting.]

The example in 10.1.7.2 [dcl.type.simple] paragraph 4 reads, in part,

```
const int&& foo();
int i;
decltype(foo()) x1 = i; // type is const int&&
```

The initialization is an ill-formed attempt to bind an rvalue reference to an lvalue.

Proposed resolution (April, 2013):

Change the example in 10.1.7.2 [dcl.type.simple] paragraph 4 as follows:

```
const int&& foo();
int i;
struct A { double x; };
const A* a = new A();
decltype(foo()) x1 = i;      // type is const int&&
decltype(i) x2;              // type is int
decltype(a->x) x3;           // type is double
decltype((a->x)) x4 = x3;    // type is const double&
```

1852. Wording issues regarding `decltype(auto)`

Section: 10.1.7.2 [dcl.type.simple] **Status:** CD4 **Submitter:** Daveed Vandevoorde **Date:** 2014-02-04

[Moved to DR at the November, 2014 meeting.]

According to 10.1.7.2 [dcl.type.simple] paragraph 2,

> The `auto` specifier is a placeholder for a type to be deduced (10.1.7.4 [dcl.spec.auto]).

This is not true when `auto` appears in the `decltype(auto)` construct.

On a slightly related wording issue, 10.1.7.4 [dcl.spec.auto] paragraph 2 says,

> The `auto` and `decltype(auto)` *type-specifier*s designate a placeholder type that will be replaced later, either by deduction from an initializer or by explicit specification with a *trailing-return-type*.

This could be read as implying that `decltype(auto)` can be used to introduce a function with a *trailing-return-type*, contradicting 11.3.5 [dcl.fct] paragraph 2, which requires that a function declarator with a *trailing-return-type* must have `auto` as the sole type specifier.

Proposed resolution (February, 2014):

1. Change 10.1.7.2 [dcl.type.simple] paragraph 2 as follows:

> The ***simple-type-specifier*** `auto` ~~specifier~~ is a placeholder for a type to be deduced (10.1.7.4 [dcl.spec.auto]). The other *simple-type-specifier*s...

2. Change 10.1.7.4 [dcl.spec.auto] paragraph 1 as follows:

> The `auto` and `decltype(auto)` *type-specifier*s **are used to** designate a placeholder type that will be replaced later~~, either~~ by deduction from an initializer ~~or by explicit specification with a *trailing-return-type*~~. The `auto` *type-specifier* is also used **to introduce a function type having a *trailing-return-type* or** to signify that a lambda is a generic lambda.

2157. Further disambiguation of enumeration *elaborated-type-specifier*

Section: 10.1.7.3 [dcl.type.elab] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2015-07-06

[Adopted at the February, 2016 meeting.]

The resolution of issue 1966 does not apply to the case where the enumeration name is qualified, e.g.,

```
enum E : int;
struct X {
  enum ::E : int();
};
```

Proposed resolution (October, 2015):

Change 10.2 [dcl.enum] paragraph 1 as follows:

> ...A `:` following "`enum` ***nested-name-specifier*$_{opt}$** *identifier*" within the *decl-specifier-seq* of a *member-declaration* is parsed as part of an *enum-base*. [*Note:* This resolves a potential ambiguity...

1877. Return type deduction from `return` with no operand

Section: 10.1.7.4 [dcl.spec.auto] **Status:** CD4 **Submitter:** Daveed Vandevoorde **Date:** 2014-02-20

[Moved to DR at the November, 2014 meeting.]

Return type deduction from a `return` statement with no *expression* is described in 10.1.7.4 [dcl.spec.auto] paragraph 7 as follows:

> When a variable declared using a placeholder type is initialized, or a `return` statement occurs in a function declared with a return type that contains a placeholder type, the deduced return type or variable type is determined from the type of its initializer. In the case of a `return` with no operand, the initializer is considered to be `void()`. Let `T` be the declared type of the variable or return type of the function. If the placeholder is the `auto` *type-specifier*, the deduced type is determined using the rules for template argument deduction. If the deduction is for a `return` statement and the initializer is a *braced-init-list* (11.6.4 [dcl.init.list]), the program is ill-formed. Otherwise, obtain `P` from `T` by replacing the occurrences of `auto` with either a new invented type template parameter `U` or, if the initializer is a *braced-init-list*, with `std::initializer_list<U>`. Deduce a value for `U` using the rules of template argument deduction from a function call (17.9.2.1 [temp.deduct.call]), where `P` is a function template parameter type and the initializer is the corresponding argument.

However, this does not work: the deduction for an argument of `void()` would give a parameter type of `void` and be ill-formed. It would be better simply to say that the deduced type in this case is `void`.

In a related example, consider

```
decltype(auto) f(void *p) {
  return *p;
}
```

This is presumably an error because `decltype(*p)` would be `void&`, which is ill-formed. Perhaps this case should be mentioned explicitly.

Notes from the June, 2014 meeting:

The last part of the issue is not a defect, because the unary `*` operator requires its operand to be a pointer to an object or function type, and `void` is neither, so the expression is ill-formed and deduction does not occur for that case.

It was also observed during the discussion that the same deduction problem occurs when returning an expression of type `void` as when the expression is omitted, so the resolution should cover both cases.

Proposed resolution (June, 2014):

Change 10.1.7.4 [dcl.spec.auto] paragraph 7 as follows:

> When a variable declared using a placeholder type is initialized, or a `return` statement occurs in a function declared with a return type that contains a placeholder type, the deduced return type or variable type is determined from the type of its initializer. In

the case of a `return` with no operand **or with an operand of type** `void`, the ~~initializer~~ **declared return type shall be** `auto` **and the deduced return type** is `void` ~~considered to be `void()`~~. ~~Let~~ **Otherwise, let** `T` be the declared type...

1878. `operator auto` **template**

Section: 10.1.7.4 [dcl.spec.auto] **Status:** CD4 **Submitter:** Daveed Vandevoorde **Date:** 2014-02-20

[Moved to DR at the November, 2014 meeting.]

The Standard currently appears to allow something like

```
struct S {
  template<class T> operator auto() { return 42; }
};
```

This is of very limited utility and presents difficulties for some implementations. It might be good to prohibit such constructs.

Proposed resolution (October, 2014):

Add the following as the last paragraph of 15.3.2 [class.conv.fct]:

> **A conversion function template shall not have a deduced return type (10.1.7.4 [dcl.spec.auto]).**

1892. Use of `auto` **in function type**

Section: 10.1.7.4 [dcl.spec.auto] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-03-12

[Moved to DR at the November, 2014 meeting.]

There appear to be no restrictions against using the `auto` specifier in examples like the following:

```
template<typename T> using X = T;
X<auto()> f_with_deduced_return_type; // ok
std::vector<auto(*)()> v; // ok?!
void f(auto (*)()); // ok?!
```

Proposed resolution (June, 2014):

Change 10.1.7.4 [dcl.spec.auto] paragraph 2 as follows:

> The placeholder type can appear with a function declarator in the *decl-specifier-seq*, *type-specifier-seq*, *conversion-function-id*, or *trailing-return-type*, in any context where such a declarator is valid. If the function declarator includes a *trailing-return-type* (11.3.5 [dcl.fct]), that specifies the declared return type of the function. **Otherwise, the function declarator shall declare a function.** If the declared return type of the function contains a placeholder type, the return type of the function is deduced from return statements in the body of the function, if any.

1958. `decltype(auto)` **with parenthesized initializer**

Section: 10.1.7.4 [dcl.spec.auto] **Status:** CD4 **Submitter:** Vinny Romano **Date:** 2014-06-30

[Moved to DR at the May, 2015 meeting.]

According to 10.1.7.4 [dcl.spec.auto] paragraph 7,

> If the placeholder is the `decltype(auto)` *type-specifier*, the declared type of the variable or return type of the function shall be the placeholder alone. The type deduced for the variable or return type is determined as described in 10.1.7.2 [dcl.type.simple], as though the initializer had been the operand of the `decltype`.

This is problematic when the parenthesized form of *initializer* is used, e.g.,

```
int i;
decltype(auto) x(i);
```

the specification would deduce the type as `decltype((i))`, or `int&`. The wording should be clarified that the *expression* and not the entire *initializer* is considered to be the operand of `decltype`.

Proposed resolution (April, 2015):

Change 10.1.7.4 [dcl.spec.auto] paragraph 7 as follows:

> ...If the placeholder is the `decltype(auto)` *type-specifier*, the declared type of the variable or return type of the function shall be the placeholder alone. The type deduced for the variable or return type is determined as described in 10.1.7.2 [dcl.type.simple],

as though the ~~initializer~~ ***initializer-clause*** **or** ***expression-list*** **of the** ***initializer*** **or the** ***expression*** **of the** `return` **statement** had been the operand of the decltype. [*Example:*

```
int i;
int&& f();
auto            x2a(i);    // decltype(x2a) is int
decltype(auto) x2d(i);    // decltype(x2d) is int
auto           x3a = i;   // decltype(x3a) is int
decltype(auto) x3d = i;   // decltype(x3d) is int
...
```

2044. `decltype(auto)` **and** `void`

Section: 10.1.7.4 [dcl.spec.auto] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-11-14

[Adopted at the February, 2016 meeting.]

The resolution of issue 1877 does not correctly handle `decltype(auto)` return types with `void` return expressions:

```
T f();
decltype(auto) g() { return f(); }
```

fails when `T` is `void`.

Suggested resolution:

Change 10.1.7.4 [dcl.spec.auto] paragraph 7 as follows:

> ...In the case of a `return` with no operand or with an operand of type `void`, the declared return type shall be `auto` **or** `decltype(auto)` and the deduced return type is `void`. Otherwise...

Proposed resolution (September, 2015):

Change 10.1.7.4 [dcl.spec.auto] paragraph 7 as follows:

> When a variable declared using a placeholder type is initialized, or a `return` statement occurs in a function declared with a return type that contains a placeholder type, the deduced return type or variable type is determined from the type of its initializer. In the case of a `return` with no operand or with an operand of type `void`**:**
>
> - **if the declared return type is** `decltype(auto)`**, then the deduced return type is** `void`**;**
> - **otherwise,** the declared return type shall be ***cv*** `auto` and the deduced return type is ***cv*** `void`.
>
> Otherwise, let `T` be...

1638. Declaring an explicit specialization of a scoped enumeration

Section: 10.2 [dcl.enum] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2013-03-12

[Adopted at the February, 2016 meeting.]

There is no syntax currently for declaring an explicit specialization of a member scoped enumeration. A declaration (not a definition) of such an explicit specialization most resembles an *opaque-enum-declaration*, but the grammar for that requires that the name be a simple identifier, which will not be the case for an explicit specialization of a member enumeration. This could be remedied by adding a *nested-name-specifier* to the grammar with a restriction that a *nested-name-specifier* only appear in an explicit specialization.

Proposed resolution (October, 2015):

1. Change the grammar in 10.2 [dcl.enum] paragraph 1 as follows:

   opaque-enum-declaration:
   enum-key attribute-specifier-seq$_{opt}$ ***nested-name-specifier***$_{opt}$ *identifier enum-base*$_{opt}$;

2. Add the following at the end of 10.2 [dcl.enum] paragraph 1:

   If an ***opaque-enum-declaration*** **contains a** ***nested-name-specifier*****, the declaration shall be an explicit specialization (17.8.3 [temp.expl.spec]).**

1766. Values outside the range of the values of an enumeration

Section: 10.2 [dcl.enum] **Status:** CD4 **Submitter:** CWG **Date:** 2013-09-23

[Moved to DR at the November, 2014 meeting.]

Although issue 1094 clarified that the value of an expression of enumeration type might not be within the range of the values of the enumeration after a conversion to the enumeration type (see 8.2.9 [expr.static.cast] paragraph 10), the result is simply an unspecified value. This should probably be strengthened to produce undefined behavior, in light of the fact that undefined behavior makes an expression non-constant. See also 12.2.4 [class.bit] paragraph 4.

Proposed resolution (February, 2014):

Change 8.2.9 [expr.static.cast] paragraph 10 as follows:

> A value of integral or enumeration type can be explicitly converted to an enumeration type. The value is unchanged if the original value is within the range of the enumeration values (10.2 [dcl.enum]). Otherwise, the ~~resulting value is unspecified (and might not be in that range)~~ **behavior is undefined**. A value of floating-point type...

1966. Colon following enumeration *elaborated-type-specifier*

Section: 10.2 [dcl.enum] **Status:** CD4 **Submitter:** Daveed Vandevoorde **Date:** 2014-07-08

[Moved to DR at the May, 2015 meeting.]

The resolution of issue 1514 was intended to apply whenever `enum` *identifier* appears; however, the fact that the disambiguation rule appears in the description of the *enum-specifier* makes it unclear that it is intended to apply in other contexts such as:

```
enum E { e1 };
void f() {
  false ? new enum E : int();
}
```

Proposed resolution (April, 2015):

Change 10.2 [dcl.enum] paragraph 1 as follows:

> A : following "`enum` *identifier*" **within the *decl-specifier-seq* of a *member-declaration*** is parsed as part of an *enum-base*. [*Note:* This resolves a potential ambiguity between the declaration of an enumeration with an *enum-base* and the declaration of an unnamed bit-field of enumeration type...

2156. Definition of enumeration declared by *using-declaration*

Section: 10.2 [dcl.enum] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2015-07-06

[Adopted at the February, 2016 meeting.]

The description of enumeration declarations in 10.2 [dcl.enum] does not, but should, contain similar wording to that preventing a class definition from defining a class type named by a *using-declaration*:

> If a *class-head-name* contains a *nested-name-specifier*, the *class-specifier* shall refer to a class that was previously declared directly in the class or namespace to which the *nested-name-specifier* refers, or in an element of the inline namespace set (10.3.1 [namespace.def]) of that namespace (i.e., not merely inherited or introduced by a *using-declaration*), and the *class-specifier* shall appear in a namespace enclosing the previous declaration. In such cases, the *nested-name-specifier* of the *class-head-name* of the definition shall not begin with a *decltype-specifier*.

Proposed resolution (February, 2016):

Add the following as a new paragraph at the end of 10.2 [dcl.enum]:

> **If an *enum-head* contains a *nested-name-specifier*, the *enum-specifier* shall refer to an enumeration that was previously declared directly in the class or namespace to which the *nested-name-specifier* refers, or in an element of the inline namespace set (10.3.1 [namespace.def]) of that namespace (i.e., not merely inherited or introduced by a *using-declaration*), and the *enum-specifier* shall appear in a namespace enclosing the previous declaration. In such cases, the *nested-name-specifier* of the *enum-head* of the definition shall not begin with a *decltype-specifier*.**

---

987. Which declarations introduce namespace members?

Section: 10.3 [basic.namespace] **Status:** CD4 **Submitter:** Michael Wong **Date:** 19 October, 2009

[Moved to DR at the November, 2014 meeting.]

According to 10.3 [basic.namespace] paragraph 1,

The name of a namespace can be used to access entities declared in that namespace; that is, the members of the namespace.

implying that all declarations in a namespace, including definitions of members of nested namespaces, explicit instantiations, and explicit specializations, introduce members of the containing namespace. 10.3.1.2 [namespace.memdef] paragraph 3 clarifies the intent somewhat:

Every name first declared in a namespace is a member of that namespace.

However, current changes to clarify the behavior of deleted functions (which must be deleted on their "first declaration") state that an explicit specialization of a function template is its first declaration.

Proposed resolution (November, 2014):

This issue is resolved by the resolution of issue 1838.

1657. Attributes for namespaces and enumerators

Section: 10.3.1 [namespace.def] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2013-08-26

[Addressed by the adoption of paper N4266 at the November, 2104 meeting.]

During the discussion of paper N3394, it was observed that the grammar does not currently, but perhaps should, permit attributes to be specified for namespaces and enumerators.

1795. Disambiguating *original-namespace-definition* and *extension-namespace-definition*

Section: 10.3.1 [namespace.def] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2013-10-04

[Moved to DR at the November, 2014 meeting.]

According to 10.3.1 [namespace.def] paragraph 2,

> The *identifier* in an *original-namespace-definition* shall not have been previously defined in the declarative region in which the *original-namespace-definition* appears.

Apparently the intent of this requirement is to say that, given the declarations

```
namespace N { }
namespace N { }
```

the second declaration is to be taken as an *extension-namespace-definition* and not an *original-namespace-definition*, since the general rules in 6.3.1 [basic.scope.declarative] cover the case in which the *identifier* has been previously declared as something other than a namespace.

This use of "shall" for disambiguation is novel, however, and it would be better to replace it with a specific statement addressing disambiguation in paragraphs 2 and 3.

Proposed Resolution (July, 2014):

1. Change 6.3.6 [basic.scope.namespace] paragraph 1 as follows:

> The declarative region of a *namespace-definition* is its *namespace-body*. ~~The potential scope denoted by an *original-namespace-name* is the concatenation of the declarative regions established by each of the *namespace-definitions* in the same declarative region with that *original-namespace-name*.~~ Entities declared in a *namespace-body*...

2. Change 10.3.1 [namespace.def] paragraphs 1-4 as follows:

> The grammar for a *namespace-definition* is
>
> *namespace-name:*
> ~~*original-namespace-name*~~ ***identifier***
> *namespace-alias*
> ~~*original-namespace-name:*~~
> ~~*identifier*~~
> *namespace-definition:*
> *named-namespace-definition*
> *unnamed-namespace-definition*
> *named-namespace-definition:*
> ~~*original-namespace-definition*~~
> ~~*extension-namespace-definition*~~
> ~~*original-namespace-definition:*~~
> `inline`$_{opt}$ `namespace` *identifier* `{` *namespace-body* `}`
> ~~*extension-namespace-definition:*~~

~~`inline`$_{opt}$ `namespace` *original-namespace-name* `{` *namespace-body* `}`~~

unnamed-namespace-definition:

`inline`$_{opt}$ `namespace { `*namespace-body*` }`

namespace-body:

declaration-seq$_{opt}$

~~The identifier in an *original-namespace-definition* shall not have been previously defined in the declarative region in which the *original-namespace-definition* appears. The identifier in an *original-namespace-definition* is the name of the namespace. Subsequently in that declarative region, it is treated as an *original-namespace-name*.~~

~~The *original-namespace-name* in an *extension-namespace-definition* shall have previously been defined in an *original-namespace-definition* in the same declarative region.~~

Every *namespace-definition* shall appear in the global scope or in a namespace scope (6.3.6 [basic.scope.namespace]).

In a *named-namespace-definition*, the *identifier* is the name of the namespace. If the *identifier*, when looked up (6.4.1 [basic.lookup.unqual]), refers to a *namespace-name* (but not a *namespace-alias*) introduced in the declarative region in which the *named-namespace-definition* appears, the *namespace-definition extends* the previously-declared namespace. Otherwise, the *identifier* is introduced as a *namespace-name* into the declarative region in which the *named-namespace-definition* appears.

3. Change 10.3.1 [namespace.def] paragraph 7 as follows:

   If the optional initial `inline` keyword appears in a *namespace-definition* for a particular namespace, that namespace is declared to be an *inline namespace*. The `inline` keyword may be used on ~~an *extension-namespace-definition*~~ **a *namespace-definition* that extends a namespace** only if it was previously used on the ~~*original-namespace-definition*~~ ***namespace-definition* that initially declared the *namespace-name*** for that namespace.

4. Delete 10.3.2 [namespace.alias] paragraph 4:

   ~~A *namespace-name* or *namespace-alias* shall not be declared as the name of any other entity in the same declarative region. A *namespace-name* defined at global scope shall not be declared as the name of any other entity in any global scope of the program. No diagnostic is required for a violation of this rule by declarations in different translation units.~~

5. Change 10.3.4 [namespace.udir] paragraph 5 as follows:

   If a namespace is extended ~~by an *extension-namespace-definition*~~ after a *using-directive* for that namespace is given, the additional members of the extended namespace and the members of namespaces nominated by *using-directive*s in the ~~*extension-namespace-definition*~~ **extending *namespace-definition*** can be used after the ~~*extension-namespace-definition*~~ **extending *namespace-definition***.

2061. Inline namespace after simplifications

Section: 10.3.1 [namespace.def] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-12-18

[Adopted at the February, 2016 meeting.]

After the resolution of issue 1795, 10.3.1 [namespace.def] paragraph 3 now says:

> In a *named-namespace-definition*, the *identifier* is the name of the namespace. If the *identifier*, when looked up (6.4.1 [basic.lookup.unqual]), refers to a *namespace-name* (but not a *namespace-alias*) introduced in the declarative region in which the *named-namespace-definition* appears, the *namespace-definition extends* the previously-declared namespace. Otherwise, the *identifier* is introduced as a *namespace-name* into the declarative region in which the *named-namespace-definition* appears.

This appears to break code like the following:

```
namespace A {
  inline namespace b {
    namespace C {
      template<typename T> void f();
    }
  }
}

namespace A {
  namespace C {
    template<> void f<int>() { }
  }
}
```

because (by definition of "declarative region") `C` cannot be used as an unqualified name to refer to `A::b::C` within `A` if its declarative region is `A::b`.

Proposed resolution (September, 2015):

Change 10.3.1 [namespace.def] paragraph 3 as follows:

In a *named-namespace-definition*, the *identifier* is the name of the namespace. If the *identifier*, when looked up (6.4.1 [basic.lookup.unqual]), refers to a *namespace-name* (but not a *namespace-alias*) **that was** introduced in the ~~declarative region~~ **namespace** in which the *named-namespace-definition* appears **or that was introduced in a member of the inline namespace set of that namespace**, the *namespace-definition* extends the previously-declared namespace. Otherwise, the *identifier* is introduced as a *namespace-name* into the declarative region in which the *named-namespace-definition* appears.

1021. Definitions of namespace members

Section: 10.3.1.2 [namespace.memdef] **Status:** CD4 **Submitter:** Michael Wong **Date:** 2010-01-14

[Moved to DR at the November, 2014 meeting.]

According to 10.3.1.2 [namespace.memdef] paragraphs 1 and 2 read,

> Members (including explicit specializations of templates (17.8.3 [temp.expl.spec])) of a namespace can be defined within that namespace.
>
> Members of a named namespace can also be defined outside that namespace by explicit qualification (6.4.3.2 [namespace.qual]) of the name being defined, provided that the entity being defined was already declared in the namespace and the definition appears after the point of declaration in a namespace that encloses the declaration's namespace.

It is not clear what these specifications mean for the following pair of examples:

```
namespace N {
    struct A;
}
using N::A;
struct A { };
```

Although this does not satisfy the "by explicit qualification" requirement, it is accepted by major implementations.

```
struct S;
namespace A {
    using ::S;
    struct S  { };
}
```

Is this a definition "within that namespace," or should that wording be interpreted as "directly within" the namespace?

See also issue 1838.

Proposed Resolution (July, 2014):

This issue is resolved by the resolution of issue 1838.

1838. Definition via *unqualified-id* and *using-declaration*

Section: 10.3.1.2 [namespace.memdef] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-01-17

[Moved to DR at the November, 2014 meeting.]

The Standard is not clear about what happens when an entity is declared but not defined in an inner namespace and declared via a *using-declaration* in an outer namespace, and a definition of an entity with that name as an *unqualified-id* appears in the outer namespace. Is this a legitimate definition of the inner-namespace entity, as it would be if the definition used a *qualified-id*, or is the definition a member of the outer namespace and thus in conflict with the *using-declaration*? There is implementation divergence on the treatment of such definitions.

See also issues 1708 and 1021.

Notes from the February, 2014 meeting:

CWG agreed that the definition in such cases is a member of the outer namespace, not a redeclaration of the name introduced in that namespace by the *using-declaration*.

Proposed Resolution (July, 2014):

1. Change 10.3.1.2 [namespace.memdef] paragraph 1 as follows:

> ~~Members (including explicit specializations of templates (17.8.3 [temp.expl.spec])) of a namespace can be defined within that namespace.~~ **A declaration in a namespace `N` (excluding declarations in nested scopes) whose *declarator-id* is an *unqualified-id* declares (or redeclares) a member of `N`, and may be a definition. [*Note:* An explicit instantiation (17.8.2 [temp.explicit]) or explicit specialization (17.8.3 [temp.expl.spec]) of a template does not introduce a name and thus may be declared using an *unqualified-id* in a member of the enclosing namespace set, if the primary template is declared in an inline namespace. —*end note*]** [*Example:*

```
namespace X {
  void f() { /* ... */ }  // OK: introduces X::f()

  namespace M {
    void g();             // OK: introduces X::M::g()
  }
  using M::g;
  void g();               // error: conflicts with X::M::g()
}
```

—*end example*]

2. Change 10.3.1.2 [namespace.memdef] paragraph 3 as follows:

> ~~Every name first declared in a namespace is a member of that namespace.~~ If a `friend` declaration...

This resolution also resolves issues 1021 and 987.

1887. Problems with `::` as *nested-name-specifier*

Section: 10.3.3 [namespace.udecl] **Status:** CD4 **Submitter:** Jeff Snyder **Date:** 2014-03-04

[Moved to DR at the November, 2014 meeting.]

Issue 1411 added `::` as a production for *nested-name-specifier*. However, the grammar for *using-declaration*s should have been updated but was overlooked:

> *using-declaration:*
> `using typename`$_{opt}$ *nested-name-specifier unqualified-id* `;`
> `using ::` *unqualified-id* `;`

In addition, there is some verbiage in 6.4.3.2 [namespace.qual] paragraph 1 and 10.3.3 [namespace.udecl] paragraph 9 that should probably be revised.

Proposed resolution (October, 2014):

1. Change the grammar in 10.3.3 [namespace.udecl] paragraph 1 as follows:

> *using-declaration:*
> `using typename`$_{opt}$ *nested-name-spaceifier unqualified-id* `;`
> ~~`using ::` *unqualified-id* `;`~~

2. Change 6.4.3.2 [namespace.qual] paragraph 1 as follows:

> If the *nested-name-specifier* of a *qualified-id* nominates a namespace **(including the case where the *nested-name-specifier* is `::`, i.e., nominating the global namespace)**, the name specified after the *nested-name-specifier* is looked up in the scope of the namespace. ~~If a *qualified-id* starts with `::`, the name after the `::` is looked up in the global namespace. In either case, the~~ **The** names in a *template-argument* of a *template-id* are looked up in the context in which the entire *postfix-expression* occurs.

3. Change _N4567_.5.1.1 [expr.prim.general] paragraph 10 as follows:

> ~~A `::`, or a~~ **The *nested-name-specifier* `::` names the global namespace. A** *nested-name-specifier* that names a namespace (10.3 [basic.namespace]), ~~in either case~~ followed by the name of a member of that namespace (or the name of a member of a namespace made visible by a *using-directive*)**,** is a *qualified-id*; 6.4.3.2 [namespace.qual] describes name lookup for namespace members that appear in *qualified-id*s. The result is...

4. Change 10.3.3 [namespace.udecl] paragraph 9 as follows:

> Members declared by a *using-declaration* can be referred to by explicit qualification just like other member names (6.4.3.2 [namespace.qual]). ~~In a *using-declaration*, a prefix `::` refers to the global namespace.~~ [*Example:*

1903. What declarations are introduced by a non-member *using-declaration*?

Section: 10.3.3 [namespace.udecl] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-03-26

[Adopted at the October, 2015 meeting as P0136R1.]

The set of declarations introduced by a *using-declaration* that is a *member-declaration* is specified by 10.3.3 [namespace.udecl] paragraph 3. However, there is no corresponding specification for a non-member *using-declaration*.

1708. overly-strict requirements for names with C language linkage

Section: 10.5 [dcl.link] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2013-06-29

[Moved to DR at the November, 2014 meeting.]

According to 10.5 [dcl.link] paragraph 6,

> An entity with C language linkage shall not be declared with the same name as an entity in global scope, unless both declarations denote the same entity; no diagnostic is required if the declarations appear in different translation units.

This restriction is too broad; it does not allow for the so-called `stat` hack, where a C-linkage function and a class are both declared in global scope, and it does not allow for function overloading, either. It should be revised to apply only to variables.

Additional note (February, 2014):

See also issue 1838 for an interaction with *using-declarations.*

Proposed resolution (February, 2014):

Change 10.5 [dcl.link] paragraph 6 as follows:

> ...An entity with C language linkage shall not be declared with the same name as ~~an entity~~ **a variable** in global scope, unless both declarations denote the same entity; no diagnostic is required if the declarations appear in different translation units...

Additional note, May, 2014:

It was observed that this resolution would allow a definition of `main` as a C-linkage variable in a namespace. The issue is being returned to "review" status for further discussion.

2079. `[[` appearing in a *balanced-token-seq*

Section: 10.6.1 [dcl.attr.grammar] **Status:** CD4 **Submitter:** Jonathan Caves **Date:** 2015-02-03

[Adopted at the February, 2016 meeting.]

According to 10.6.1 [dcl.attr.grammar] paragraph 6,

> Two consecutive left square bracket tokens shall appear only when introducing an *attribute-specifier*. [*Note:* If two consecutive left square brackets appear where an *attribute-specifier* is not allowed, the program is ill-formed even if the brackets match an alternative grammar production. —*end note*]

In order to allow program fragments to appeae within attributes, this restriction should not apply within the *balanced-token-seq* of an attribute.

Proposed resolution (September, 2015):

Change 10.6.1 [dcl.attr.grammar] paragraph 6 as follows:

> Two consecutive left square bracket tokens shall appear only when introducing an *attribute-specifier* **or within the *balanced-token-seq* of an *attribute-argument-clause***. [*Note:* If two consecutive left square brackets appear where an *attribute-specifier* is not allowed, the program is ill-formed even if the brackets match an alternative grammar production. —*end note*] [*Example:*
>
> ```
> int p[10];
> void f() {
>   int x = 42, y[5];
>   int(p[[x] { return x; }()]);  // error: invalid attribute on a nested
>                                 // declarator-id and not a function-style cast of
>                                 // an element of p.
>   y[[] { return 2; }()] = 2;    // error even though attributes are not allowed
>                                 // in this context.
>
>   int i [[vendor::attr([[]])]]; // well-formed implementation-defined attribute.
> }
> ```
>
> —*end example*]

1615. Alignment of types, variables, and members

Section: 10.6.2 [dcl.align] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2013-02-01

[Moved to DR at the November, 2014 meeting.]

According to 10.6.2 [dcl.align] paragraph 5,

The combined effect of all *alignment-specifier*s in a declaration shall not specify an alignment that is less strict than the alignment that would be required for the entity being declared if all *alignment-specifier*s were omitted (including those in other declarations).

Presumably the intent was "other declarations of the same entity," but the wording as written could be read to make the following example well-formed (assuming `alignof(int)` is 4):

```
struct alignas(4) A {
  alignas(8) int n;
};
struct alignas(8) B {
  char c;
};
alignas(1) B b;
struct alignas(1) C : B {};
enum alignas(8) E : int { k };
alignas(4) E e = k;
```

Proposed resolution (February, 2014):

Change 10.6.2 [dcl.align] paragraph 5 as follows:

...if all *alignment-specifier*s **appertaining to that entity** were omitted ~~(including those in other declarations)~~. **[*Example:***

```
struct alignas(8) S {};
struct alignas(1) U {
  S s;
};   // Error: U specifies an alignment that is less strict than
     // if the alignas(1) were omitted.
```

***—end example*]**

2027. Unclear requirements for multiple `alignas` specifiers

Section: 10.6.2 [dcl.align] **Status:** CD4 **Submitter:** Steve Clamage **Date:** 2014-10-20

[Moved to DR at the October, 2015 meeting.]

The description of *alignment-specifier*s is unclear. For example, 10.6.2 [dcl.align] bullet 2.2 says,

> if the constant expression evaluates to a fundamental alignment, the alignment requirement of the declared entity shall be the specified fundamental alignment

However, paragraph 4 says,

> When multiple *alignment-specifier*s are specified for an entity, the alignment requirement shall be set to the strictest specified alignment.

meaning that a less-strict alignment will be ignored, rather than being the alignment of the entity, and presumably meaning that no diagnostic is required for an insufficiently-strict alignment if a more stringent requirement is also supplied.

Proposed resolution (May, 2015):

1. Change 10.6.2 [dcl.align] paragraph 2 as follows:

   When the *alignment-specifier* is of the form `alignas(` *constant-expression* `)`:

   - the *constant-expression* shall be an integral constant expression**;**
   - ~~if the constant expression evaluates to a fundamental alignment, the alignment requirement of the declared entity shall be the specified fundamental alignment~~
   - ~~if the constant expression evaluates to an extended alignment and the implementation supports that alignment in the context of the declaration, the alignment of the declared entity shall be that alignment~~
   - if the constant expression **does not evaluate to an alignment value (6.11 [basic.align]), or** evaluates to an extended alignment and the implementation does not support that alignment in the context of the declaration, the program is ill-formed**.**
   - ~~if the constant expression evaluates to zero, the alignment specifier shall have no effect~~
   - ~~otherwise, the program is ill-formed.~~

2. Change 10.6.2 [dcl.align] paragraph 4 as follows:

   ~~When multiple *alignment-specifier*s are specified for an entity, the~~ **The** alignment requirement ~~shall be set to~~ **of an entity is** the strictest ~~specified~~ **non-zero** alignment **specified by its *alignment-specifier*s, if any; otherwise, the *alignment-specifier*s have no effect**.

2040. *trailing-return-type* no longer ambiguous

Section: 11 [dcl.decl] **Status:** CD4 **Submitter:** Jens Maurer **Date:** 2014-11-09

[Adopted at the February, 2016 meeting.]

According to 11 [dcl.decl] paragraph 5,

> The *type-id* in a *trailing-return-type* includes the longest possible sequence of *abstract-declarator*s. [*Note:* This resolves the ambiguous binding of array and function declarators. [*Example:*
>
> ```
> auto f()->int(*)[4]; // function returning a pointer to array[4] of int
>                      // not function returning array[4] of pointer to int
> ```
>
> —*end example*] —*end note*]

However, the grammar has changed since that rule and example were added; because *trailing-return-type* can only appear at the top level, there is no longer any potential ambiguity.

Proposed resolution (September, 2015):

Delete 11 [dcl.decl] paragraph 5:

> ~~The optional *attribute-specifier-seq* in a *trailing-return-type* appertains to the indicated return type. The *type-id* in a *trailing-return-type* includes the longest possible sequence of *abstract-declarator*s. [*Note:* This resolves the ambiguous binding of array and function declarators. [*Example:*~~
>
> ~~`auto f()->int(*)[4]; // function returning a pointer to array[4] of int`~~
> ~~`// not function returning array[4] of pointer to int`~~
>
> ~~—*end example*] —*end note*]~~

2175. Ambiguity with attribute in conversion operator declaration

Section: 11.2 [dcl.ambig.res] **Status:** CD4 **Submitter:** Daveed Vandevoorde **Date:** 2015-09-17

[Adopted at the February, 2016 meeting.]

The declaration

```
operator int [[noreturn]] ();
```

is ambiguous with respect to the binding of the attribute. It could either be parsed (as apparently intended by the user) as part of the *noptr-declarator* (11 [dcl.decl] paragraph 4) or as part of the *type-specifier-seq* (10.1.7 [dcl.type] paragraph 1) of the *conversion-type-id* (15.3.2 [class.conv.fct] paragraph 1). Current implementations disambiguate this declaration in favor of the latter interpretation, issuing an error for the declaration because the `noreturn` attribute cannot apply to a type.

Proposed resolution (February, 2016):

Change 15.3.2 [class.conv.fct] paragraph 3 as follows:

> The *conversion-type-id* shall not represent a function type nor an array type. The *conversion-type-id* in a *conversion-function-id* is the longest ~~possible~~ sequence of ~~*conversion-declarator*s~~ **tokens that could possibly form a *conversion-type-id***. [*Note:* This prevents ambiguities between the declarator operator `*` and its expression counterparts. [*Example:*
>
> ```
> &ac.operator int*i;  // syntax error:
>                      // parsed as: &(ac.operator int *)i
>                      // not as: &(ac.operator int)*i
> ```
>
> The `*` is the pointer declarator and not the multiplication operator. —*end example*] **This rule also prevents ambiguities for attributes. [*Example:***
>
> ```
> operator int [[noreturn]] (); // error: noreturn attribute applied to a type
> ```
>
> ***—end example***] —*end note*]

2113. Incompete specification of types for declarators

Section: 11.3 [dcl.meaning] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2015-04-08

[Adopted at the February, 2016 meeting.]

According to 11.3 [dcl.meaning]

> A `static`, `thread_local`, `extern`, `register`, `mutable`, `friend`, `inline`, `virtual`, or `typedef` specifier applies directly to each *declarator-id* in an *init-declarator-list*, the type specified for each *declarator-id* depends on both the *decl-specifier-seq* and its *declarator*.

This list is missing `constexpr` and `explicit`. Also, this should apply, but doesn't, to *member-declarator-list*s.

Proposed resolution (September, 2015):

Change 11.3 [dcl.meaning] paragraph 2 as follows:

> A `static`, `thread_local`, `extern`, `register`, `mutable`, `friend`, `inline`, `virtual`, `constexpr`, `explicit`, or `typedef` specifier applies directly to each *declarator-id* in an *init-declarator-list* **or *member-declarator-list***, the type specified for each *declarator-id* depends on both the *decl-specifier-seq* and its *declarator*.

2099. Inferring the bound of an array static data member

Section: 11.3.4 [dcl.array] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2015-03-15

[Adopted at the February, 2016 meeting.]

According to 11.3.4 [dcl.array] paragraph 3,

> An array bound may also be omitted when the declarator is followed by an *initializer* (11.6 [dcl.init]). In this case the bound is calculated from the number of initial elements...

However, the grammar for *member-declarator* uses *brace-or-equal-initializer*, not *initializer*, so the following is ill-formed:

```
struct X {
  static constexpr int arr[] = { 1, 2, 3 };
};
```

Proposed resolution (October, 2015):

Change 11.3.4 [dcl.array] paragraph 3 as follows:

> ...An array bound may also be omitted when the declarator is followed by an *initializer* (11.6 [dcl.init]) **or when a declarator for a static data member is followed by a *brace-or-equal-initializer* (12.2 [class.mem])**. In ~~this case~~ **both cases** the bound is calculated from the number of initial elements...

---

393. Pointer to array of unknown bound in template argument list in parameter

Section: 11.3.5 [dcl.fct] **Status:** CD4 **Submitter:** Mark Mitchell **Date:** 12 Dec 2002

[Moved to DR at the November, 2014 meeting.]

EDG rejects this code:

```
template <typename T>
struct S {};

void f (S<int (*)[]>);
```

G++ accepts it.

This is another case where the standard isn't very clear:

The language from 11.3.5 [dcl.fct] is:

> If the type of a parameter includes a type of the form "pointer to array of unknown bound of T" or "reference to array of unknown bound of T," the program is ill-formed.

Since "includes a type" is not a term defined in the standard, we're left to guess what this means. (It would be better if this were a recursive definition, the way a type theoretician would do it:

- Every type includes itself.
- T* includes T
- T[] includes T
- ...

)

Notes from April 2003 meeting:

We agreed that the example should be allowed.

Additional note (January, 2013):

Additional discussion of this issue has arisen . For example, the following is permissible:

```
T (*p) [] = (U(*)[])0;
```

but the following is not:

```
template<class T>
void sp_assert_convertible( T* ) {}

sp_assert_convertible<T[]>( (U(*)[])0 );
```

Proposed resolution (February, 2014):

Change 11.3.5 [dcl.fct] paragraph 8 as follows, including deleting the footnote:

> ~~If the type of a parameter includes a type of the form "pointer to array of unknown bound of T" or "reference to array of unknown bound of T," the program is ill-formed.~~[101] Functions shall not have a return type of type array or function, although...

1824. Completeness of return type vs point of instantiation

Section: 11.3.5 [dcl.fct] **Status:** CD4 **Submitter:** Steve Clamage **Date:** 2013-12-19

[Moved to DR at the November, 2014 meeting.]

Consider the following example:

```
template<typename T> struct A {
  T t;
};
struct S {
  A<S> f() { return A<S>(); }
};
```

According to 11.3.5 [dcl.fct] paragraph 9,

> The type of a parameter or the return type for a function definition shall not be an incomplete class type (possibly cv-qualified) unless the function is deleted (11.4.3 [dcl.fct.def.delete]) or the definition is nested within the *member-specification* for that class (including definitions in nested classes defined within the class).

Thus type `A<S>` must be a complete type. The requirement for a complete type triggers the instantiation of the template, which requires that its template argument be complete in order to use it as the type of a non-static data member.

According to 17.7.4.1 [temp.point] paragraph 4, the point of instantiation of `A<S>` is "immediately preced[ing] the namespace scope declaration or definition that refers to the specialization." Thus the point of instantiation precedes the definition of `S`, making this example ill-formed. Most or all current implementations accept the example, however.

Perhaps the specification in 11.3.5 [dcl.fct] ought to say that the completeness of the type is checked from the context of the function body (at which S is a complete type)?

Proposed resolution (February, 2014):

Change 11.3.5 [dcl.fct] paragraph 9 as follows:

> Types shall not be defined in return or parameter types. The type of a parameter or the return type for a function definition shall not be an incomplete class type (possibly cv-qualified) **in the context of the function definition** unless the function is deleted (11.4.3 [dcl.fct.def.delete]) ~~or the definition is nested within the *member-specification* for that class (including definitions in nested classes defined within the class)~~.

1814. Default arguments in *lambda-expression*s

Section: 11.3.6 [dcl.fct.default] **Status:** CD4 **Submitter:** Jonathan Caves **Date:** 2013-11-21

[Moved to DR at the November, 2014 meeting.]

The resolution for issue 974 permitting default arguments in *lambda-expression*s overlooked 11.3.6 [dcl.fct.default] paragraph 3:

> A default argument shall be specified only in the *parameter-declaration-clause* of a function declaration or in a *template-parameter* (17.1 [temp.param])...

Proposed resolution (February, 2014):

Change 11.3.6 [dcl.fct.default] paragraph 3 as follows:

> A default argument shall be specified only in the *parameter-declaration-clause* of a function declaration **or *lambda-declarator*** or in a *template-parameter* (17.1 [temp.param]); in the latter case, the *initializer-clause* shall be an *assignment-expression*. A

default argument shall not be specified for a parameter pack. If it is specified in a *parameter-declaration-clause*, it shall not occur within a declarator or *abstract-declarator* of a *parameter-declaration*.[103]

2082. Referring to parameters in unevaluated operands of default arguments

Section: 11.3.6 [dcl.fct.default] **Status:** CD4 **Submitter:** Faisal Vali **Date:** 2015-02-09

[Adopted at the February, 2016 meeting.]

According to 11.3.6 [dcl.fct.default] paragraph 9,

> A default argument is evaluated each time the function is called with no argument for the corresponding parameter. The order of evaluation of function arguments is unspecified. Consequently, parameters of a function shall not be used in a default argument, even if they are not evaluated.

This prohibits use of parameters in unevaluated operands, e.g.,

```
void foo(int a = decltype(a){});
```

This wording predates the concept of "unevaluated operands" (the phrase "not evaluated" refers to calls to the function where an actual argument is supplied and thus the default argument is not used, not to unevaluated operands) and should not apply to such cases.

Proposed resolution (October, 2015):

1. Change 11.3.6 [dcl.fct.default] paragraph 7 as follows:

> ~~Local variables~~ **A local variable** shall not ~~be used~~ **appear as a potentially-evaluated expression** in a default argument. [*Example:*
>
> ```
> void f() {
>   int i;
>   extern void g(int x = i);           // error
>    extern void h(int x = sizeof(i));    // OK
>   // ...
> }
> ```
>
> —*end example*]

2. Change 11.3.6 [dcl.fct.default] paragraph 8 as follows:

> **[*Note:*** The keyword `this` ~~shall~~ **may** not ~~be used~~ **appear** in a default argument of a member function**; see _N4567_.5.1.1 [expr.prim.general]**. [*Example:*
>
> ```
> class A {
>   void f(A* p = this) { } // error
> };
> ```
>
> —*end example*] ***—end note*]**

3. Change 11.3.6 [dcl.fct.default] paragraph 9 as follows:

> A default argument is evaluated each time the function is called with no argument for the corresponding parameter. ~~The order of evaluation of function arguments is unspecified. Consequently, parameters of a function shall not be used in a default argument, even if they are not evaluated.~~ **A parameter shall not appear as a potentially-evaluated expression in a default argument.** Parameters of a function declared before a default argument are in scope and can hide namespace and class member names. [*Example:*
>
> ```
> int a;
> int f(int a, int b = a);          // error: parameter a
>                                   // used as default argument
> typedef int I;
> int g(float I, int b = I(2));     // error: parameter I found
> int h(int a, int b = sizeof(a)); // ~~error, parameter a used~~ OK, unevaluated operand
>                                   // ~~in default argument~~
> ```
>
> —*end example*] ~~Similarly, a~~ **A** non-static member shall not ~~be used~~ **appear** in a default argument~~, even if it is not evaluated,~~ unless it appears as the *id-expression* of a class member access expression (8.2.5 [expr.ref]) or unless it is used to form a pointer to member (8.3.1 [expr.unary.op]). [*Example:*...

1791. Incorrect restrictions on *cv-qualifier-seq* and *ref-qualifier*

Section: 11.4.1 [dcl.fct.def.general] **Status:** CD4 **Submitter:** David Krauss **Date:** 2013-10-01

[Moved to DR at the November, 2014 meeting.]

Paragraph 5 of 11.4.1 [dcl.fct.def.general] says,

A *cv-qualifier-seq* or a *ref-qualifier* (or both) can be part of a non-static member function declaration, non-static member function definition, or pointer to member function only (11.3.5 [dcl.fct]); see 12.2.2.1 [class.this].

This is redundant with the specification in 11.3.5 [dcl.fct] paragraph 6 and is factually incorrect, since the list there contains other permissible constructs. It should be at most a note or possibly removed altogether.

Proposed resolution (February, 2014):

Change 11.4.1 [dcl.fct.def.general] paragraph 5 as follows:

~~A *cv-qualifier-seq* or a *ref-qualifier* (or both) can be part of a non-static member function declaration, non-static member function definition, or pointer to member function only (11.3.5 [dcl.fct]); see 12.2.2.1 [class.this].~~ **[*Note:* a *cv-qualifier-seq* affects the type of `this` in the body of a member function; see 11.3.2 [dcl.ref]. —*end note*]**

2145. Parenthesized declarator in function definition

Section: 11.4.1 [dcl.fct.def.general] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2015-06-19

[Adopted at the June, 2016 meeting.]

According to 11.4.1 [dcl.fct.def.general] paragraph 2,

> The *declarator* in a *function-definition* shall have the form
>
> `D1 (` *parameter-declaration-clause* `)` *cv-qualifier-seq*$_{opt}$
> *ref-qualifier*$_{opt}$ *exception-specification*$_{opt}$ *attribute-specifier-seq*$_{opt}$ *trailing-return-type*$_{opt}$

However, in practice implementations accept a parenthesized declarator in a function definition.

Proposed resolution (April, 2016):

Change 11.4.1 [dcl.fct.def.general] paragraph 2 as follows:

~~The *declarator* in~~ **In** a *function-definition* ~~shall have the form~~**,**

~~`D1` *parameters-and-qualifiers trailing-return-type*$_{opt}$~~

either `void` ***declarator*** `;` **or** ***declarator*** `;` **shall be a well-formed function declarator** as described in 11.3.5 [dcl.fct]. A function shall be defined only in namespace or class scope.

1552. *exception-specification*s and defaulted special member functions

Section: 11.4.2 [dcl.fct.def.default] **Status:** CD4 **Submitter:** Daveed Vandevoorde **Date:** 2012-09-07

[Moved to DR at the November, 2014 meeting.]

The current wording of 11.4.2 [dcl.fct.def.default] paragraph 2 has some surprising implications:

> An explicitly-defaulted function may be declared `constexpr` only if it would have been implicitly declared as `constexpr`, and may have an explicit *exception-specification* only if it is compatible (18.4 [except.spec]) with the *exception-specification* on the implicit declaration.

In an example like

```
struct A {
  A& operator=(A&);
};
A& A::operator=(A&) = default;
```

presumably the *exception-specification* of `A::operator=(A&)` is `noexcept(false)`. However, attempting to make that *exception-specification* explicit,

```
A& A::operator=(A&) noexcept(false) = default;
```

is an error. Is this intentional?

Proposed resolution (February, 2014):

Change 18.4 [except.spec] paragraph 4 as follows:

> ...If any declaration of a pointer to function, reference to function, or pointer to member function has an *exception-specification*, all occurrences of that declaration shall have a compatible *exception-specification*. **If a declaration of a function has an implicit *exception-specification*, other declarations of the function shall not specify an *exception-specification*.** In an explicit instantiation...

(This resolution also resolves issue 1492.)

Additional note (January, 2013):

The resolution conflicts with the current specification of `operator delete`: in 6.7.4 [basic.stc.dynamic] paragraph 2, the two `operator delete` overloads are declared with an implicit exception specification, while in 21.6 [support.dynamic] paragraph 1, they are declared as `noexcept`.

Additional note (February, 2014):

The overloads cited in the preceding note have been independently changed in N3936 to include a `noexcept` specification, making the proposed resolution correct as it stands.

1846. Declaring explicitly-defaulted implicitly-deleted functions

Section: 11.4.2 [dcl.fct.def.default] **Status:** CD4 **Submitter:** Richard Smith **Date:** 2014-01-30

[Moved to DR at the November, 2014 meeting.]

According to 11.4.2 [dcl.fct.def.default] paragraph 2,

> An explicitly-defaulted function may be declared `constexpr` only if it would have been implicitly declared as `constexpr`.

However, the rules for determining whether a function is `constexpr` and its *exception-specification* depend on the definition of function and do not apply to deleted functions. Can an explicitly-defaulted implicitly-deleted function be declared `constexpr` or have an *exception-specification*, and if so, how is its correctness to be determined?

Proposed resolution (February, 2014):

Change 11.4.2 [dcl.fct.def.default] paragrap