Issues Found Implementing C++0x

And What We Did About Them

Authors: Sean Hunt, Richard Smith, Sebastian Redl, David Majnemer
Context: Doug Gregor <doug.gregor@gmail.com>
Document number: N3307=11-0077
Date: 2011-09-08

1. **(Sean Hunt) [except.spec](15.4)/14: Functions might not actually be "directly invoked" by the definition of an implicit member.**
   <not a defect>
2. **(Sean Hunt) [class.copy](12.8)/7: Adding new special member functions to a class via default arguments**
   If a definition is used to add a default argument to a constructor, this can have disastrous effects on implicit special member functions of classes that inherit from or include an object of the first class. In particular, this can alter the overload selection and thus the implicit exception specification and triviality(!) of the implicit member function in between its declaration and definition.

   Moreover, it is not entirely clear when an implementation is required to perform the overload resolution steps to determine if a default constructor is defined as deleted or not; immediately on the completion of the class or upon the constructor's first use - clang, for instance, defers creating the declaration of an implicit member function and thus this overload resolution until it would affect the semantics of the program.

   Example:

   ```
   struct X {
     X(const X&);
   };
   struct Y {
     X x;
   };
   // Here Y would have a deleted default constructor if overload resolution
   // is performed aggressively.
   X::X(const X& = X()) { }
   // But if it were defined here, Y would have a valid default constructor.
   ```

   The suggested resolution is to make this illegal for special members only; there is no reason in principle that default arguments could not be disallowed on out-of-line definitions of member functions generally (they are already forbidden for templates), as such code is rarely useful.

   clang behavior: This case has a warning that is on by default. No significant effort is being made to ensure correct behavior one way or another when it occurs.

   suggested resolution: Add a new paragraph to [dcl.fct.default](8.3.6) reading: A default argument shall not be added to a constructor in a definition after

its declaration such that it becomes a sort of special member function (12) that it was not previously. [*Example:*

```
struct X {
  X();
  X(const X&);
};
X::X(const X& = X()) { } // ill-formed: makes X::X(const X&) into a default
                         // constructor.
```

— end example]

Delete [basic.def.odr](3.2)/5 as this paragraph is now redundant since it can only trigger on an ill-formed program.

Change [class.copy](12.8)/7 by removing the last sentence, so that it reads:

If the class definition does not explicitly declare a copy constructor, one is declared implicitly. If the class definition declares a move constructor or move assignment operator, the implicitly declared copy constructor is defined as deleted; otherwise, it is defined as defaulted (8.4). The latter case is deprecated if the class has a user-declared copy assignment operator or a user-declared destructor. Thus, for the class definition

```
struct X {
  X(const X&, int);
};
```

a copy constructor is implicitly-declared. ~~If the user-declared constructor is later defined as~~

```
~~X::X(const X& x, int i =0) { /* ... */ }~~
```

~~then any use of X's copy constructor is ill-formed because of the ambiguity; no diagnostic is required.~~

priority: Medium - This is an uncommon corner case, but getting direction from the committee in this area would reduce divergence.

3. (Sean Hunt) [except.spec](15.4)/14: Should an implicit non-throwing exception specification be dynamic or not?

In the event that an implicit member (or any destructor!) is to get an implicit non-throwing exception specification, it is not clear whether it is a noexcept specification or a throw() one. This has visible effects as it changes the outcome of violating the specification. See also Core Issue 1073, which gracefully discovered this issue and failed to address it.

clang behavior: noexcept(true) is considered stricter than throw(). This means that noexcept(true) is chosen by default, but if any implicitly called function is declared throw(), the calculated exception specification is throw(). This is overly complicated; it would be simpler to always use noexcept(true).

suggested resolution: Change [except.spec](15.4)/14 to read:
An implicitly declared special member function (Clause 12) 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 *as if with a noexcept-specification* if every function it directly invokes allows no exceptions.

priority: Low - The potential for divergence in implementations is high, but this is tempered by the fact that this difference is only observable in programs that use std::set_unexpected() and where one of these non-throwing exception specifications is violated.

4. **(Sean Hunt) [class.base.init](12.6.2)/8: Anonymous unions are initialized**

   This paragraph applies also to anonymous union members, although their members are more or less treated as a part of the enclosing structure for initialization purposes. This can lead to particularly awkward situations where a member of an anonymous union cannot be default-constructed, so the anonymous union constructor is deleted despite the fact that the member might be validly initialized by the enclosing class. A conforming implementation would have to reject this:

   ```
   struct X {
     union { const int i; }
     X() : i(0) { }
   };
   ```

   clang behavior: Unknown.

   suggested resolution: Never initialize anonymous union members directly, and instead only initialize their members.

   priority: Low - This issue appears to have existed since C++98 but has an obvious resolution and is likely not a cause for divergence, although some implementations may implement the looser, seemingly-correct behavior.

5. **(Richard Smith) [dcl.spec.auto](7.1.6.4)/6: parenthesized initializers and the auto specifier**

   This paragraph does not specify how to handle an initializer of the form (expression-list). For instance:

   ```
   auto a(1, 2, 3);
   ```

   In this case, the initializer is not an expression, so it is not meaningful to use it as a function argument. [dcl.init]/13 cannot be applied here, since we do not know whether the entity being initialized has class type.

   clang behavior: If the expression-list contains a single expression, that expression is used as the argument in the hypothetical function call. Otherwise, the declaration is ill-formed.

   suggested resolution: Change [dcl.spec.auto](7.1.6.4)/6 as follows:
   [...] The type deduced for the variable d is then the deduced A determined using the rules of template argument deduction from a function call (14.8.2.1), where P is a function template parameter type and the ~~initializer~~ *initializer-clause, braced-init-list* or single expression in the *expression-list* in d's *initializer* is the corresponding argument. If the *initializer* is a parenthesized *expression-list* which is not a single expression, or the deduction fails, the declaration is ill-formed. [...]

   priority: Low - implementations cannot and do not accept such cases anyway.

6. **(Richard Smith) [dcl.spec.auto](7.1.6.4)/7: 'auto' can represent different types for different declarators in the same declaration**

   The intent of p7 appears to be that 'auto' must represent the same type in each declarator in the same declaration, but this is violated by the handling of initializer-lists and trailing-return-types. This is well-formed:

```
  auto a = 0, b = { 1, 2, 3 }, (*f)() -> double;
```

The ‘auto’ specifier stands in for ‘int’, ‘std::initializer_list<int>’ and ‘double’, respectively.

clang behavior: Clang implements the letter of the standard for function declarations with trailing-return-types, and does not yet implement initializer-lists.

suggested resolution: Change [dcl.spec.auto]p7:
If the list of declarators contains more than one declarator, the type of each declared variable is determined as described above. For each such entity, the automatic type is the trailing-return-type if the declarator is a function declarator, std::initializer_list<V> if the initializer is a braced-init-list, and V otherwise, where V is the type deduced for the template parameter U. If the automatic type ~~deduced for the template parameter U~~ is not the same in each case, the program is ill-formed.

priority: High - gcc and clang currently follow the letter of the standard, and changing this will incur a backwards-compatibility cost if deferred.

7. **(Richard Smith) [dcl.spec.auto](7.1.6.4)/3: is 'auto' legal in a trailing-return-type?**

Is this legal?

```
int f();
auto (*g)() = &f;          // legal.
auto (*h)() -> auto = &f; // illegal?
```

The first 'auto' in the declaration of 'h' is allowed by [dcl.spec.auto] p2. The second 'auto' may be legal too, depending on the interpretation of [dcl.spec.auto]p3. It says "auto shall appear as one of the decl-specifiers in the decl-specifier-seq", which it does. However, it's a different instance of the 'auto' keyword, and the intent seems to be to reject it.

clang behavior: 'h' is rejected.

suggested resolution: Change [dcl.spec.auto]p3:
[...] This use of auto shall appear as one of the decl-specifiers in the decl-specifier-seq[...]

priority: Medium - gcc and clang already reject this, but the standard is at best unclear.

8. **(Richard Smith) [temp.alias](14.5.7]: alias template redeclarations are not required to specify an equivalent type**

This appears to be well-formed:

```
template<typename T, typename U> using A = T;
template<typename T, typename U> using A = U;
```

Since an alias template is a declaration, not a definition, the ODR does not apply to alias templates. Is [dcl.typedef]p3 intended to apply here?

clang behavior: In an alias template redeclaration, the type is required to be equivalent to the original declaration.

priority: Medium - clang already rejects such cases, but this is in defiance of the standard.

9. **(Richard Smith) [temp.alias](14.5.7)/3: This paragraph is redundant**

Since the point of declaration for an alias template was moved to the semicolon, this paragraph no longer applies. It is impossible for an alias template to refer to itself, except by (harmlessly) referring to a previous declaration.
The example in this paragraph was ill-formed without this rule (and with the old point of declaration): in the declaration of A::U, A<T> is the current instantiation, and it does not yet contain a member named U.

clang behavior: This paragraph is not implemented.

suggested resolution: Delete this paragraph.

priority: Low

10. **(Richard Smith) [dcl.typedef](7.1.3)/2: identifier in an alias-**

declaration cannot appear in the type-id

This paragraph says "The *identifier* following the using keyword [...] has the same semantics as if it were introduced by the typedef specifier. In particular, it [...] shall not appear in the *type-id*." This is incorrect: typedefs have no such restriction. This is legal:

```
typedef int A;
namespace N {
  typedef A A; // 1
  typedef A A; // 2
}
```

Both (1) and (2) would be ill-formed if they were rewritten as alias-declarations according to this wording. This is not the same semantics as a typedef specifier!

clang behavior: This restriction is not implemented.

suggested resolution: Change this paragraph as follows:
[...] 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*~~.

priority: Low - The deleted text does not appear to be intended to be normative.

11. **(Richard Smith) [class.mem](9.2): bitfield members cannot have in-class initializers**

The grammar does not allow a brace-or-equal-initializer for a bitfield member. This seems like an oversight. A brace-or-equal-initializer after a constant-expression appears to be unambiguous.

clang behavior: clang implements the letter of the standard.

suggested resolution: Change the grammar as follows:
member-declarator:
identifier$_{opt}$ *attribute-specifier-seq*$_{opt}$ *: constant-expression brace-or-equal-initializer*$_{opt}$

priority: Low - this is a backward-compatible change

12. **(Richard Smith) [class.inhctor](12.9)/3: incorrect exception specification for inherited constructors**

If the default constructor or brace-or-equal-initializer for a non-static data member can throw an exception, then the exception specification used for an inherited constructor can be tighter than the set of exceptions it may actually throw.

clang behavior: clang implements the letter of the standard. The exception specification is copied from the inherited constructor.

suggested resolution: Remove third bullet 'the exception-specification (15.4),' from [class.inhctor]/2, and change [except.spec](15.4)/14 as follows:
An implicitly declared special member function or inherited constructor (Clause 12) shall have an exception-specification. If f is an inherited 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.[...]

priority: High - implementations are currently implicitly generating code which surprisingly fails at runtime

13. **(Richard Smith) [except.spec](15.4)/14: implicit exception-specification includes too many, and too few, exceptions**

This paragraph says: "[an] 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."
However, such a function can throw exceptions by other means than a function call:

```
bool cond;
struct S {
  int n = cond ? throw "eep" : 0;
};
struct T {
  T(int n = cond ? throw "eep" : 0) noexcept;
};
struct U {
```

```
  T t;
};
```

S::S() and U::U() are given no-throw exception specifications by the current rules. Conversely, the term ‘directly invoked’ is not defined in the standard, and the only references to invocation do not indicate the invocation need be potentially-evaluated:

```
int f();
struct V {
  int n = sizeof(f());
};
```

It is unclear whether V::V() is given a noexcept(false) specification.

clang behavior: clang considers an implicit constructor to throw the exception E if either E is allowed by the chosen constructor of a base class or non-static data member with no brace-or-equal-initializer. If any brace-or-equal-initializer is not no-throwing (as determined by the rules for a noexcept expression), the constructor is implicitly noexcept(false) [clang does not track which exceptions a brace-or-init-initializer may throw]. clang does not yet consider default argument expressions used within an implicit definition.

suggested resolution: Extend [expr.unary.noexcept] to define a set of potentially-thrown exceptions for an expression, and define the exception specification for an implicit member function to be the union of those sets for each full-expression directly contained within the implicit definition of the constructor.

priority: High - implementations are currently implicitly generating code which surprisingly fails at runtime

14. (Richard Smith) ordering issues with implicit exception specification and class components with deferred parsing

In the contexts where the class is considered complete before we finish parsing the definition (function definitions, member initializers, default arguments and exception specifications), the result of parsing can depend on the exception specification of the implicit default constructor. However, the exception specification of the implicit default constructor can also depend on the results of that parse. This leads to unparseable constructs such as this:

```
template struct ExceptionIf { static int f(); };
template<> struct ExceptionIf { typedef int f; };
struct S {
  int n = ExceptionIf<noexcept(S())>::f();
};
```

Here, S::S is noexcept iff n's initializer is noexcept, which happens iff S::S is not noexcept. We can observe the same problem in the other forms in which the class is prematurely considered complete. In exception specifications:

```
struct NoExcept {
  struct T {
    T() noexcept(!noexcept(NoExcept()));
  } t;
};
```

In default arguments:

```
struct Default {
   struct T {
     T(int = ExceptionIf<noexcept(Default())>::f());
   } t;
 };
```

clang behavior: If a brace-or-equal-initializer for a non-static data member uses the exception specification of a defaulted default constructor for the class being defined, or any lexically-enclosing class within which that class is nested, the program is ill-formed. Clang does not yet support deferred parsing of default arguments and exception specifications, but the same resolution will probably be applied in those cases.

priority: High - The standard as specified is unimplementable, and there is a risk of implementations diverging on how they resolve this issue.

15. (Richard Smith) deferred parsing rules inconsistent

There are three distinct sets of rules which govern behaviour within the 'deferred' parts of a class (member function definitions, default arguments, initializers for non-static data members and exception specifications).

[class.mem]/2 says: 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).

[basic.scope.class]/1.1 says: 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, brace-or-equal-initializers of non-static data members, and default arguments in that class (including such things in nested classes).

... but not exception specifications. That's difficult to implement: the class is complete but name lookup is restricted to those declarations declared earlier. But wait...

[basic.scope.class]/1.5 says: 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 [...] member function definitions (including [...] any portion of the declarator part of such definitions which follows the declarator-id.

(Editorial note: this paragraph contains two open parentheses with no matching close parentheses).

So the potential scope of class members does includes exception specifications, default arguments and trailing return types, and also argument types! But only for member function definitions, and not for member function declarations!

[basic.lookup.unqual]/7 says: A name used in the definition of a class X outside of a member function body or nested class definition shall be declared in one of the following ways:
  - before its use [...]

Therefore name lookup in default arguments, brace-or-equal-initializers and exception specifications does not find later-declared class members, unless it's in a nested class. The next paragraph flat-out contradicts this:

[basic.lookup.unqual]/8 says: A name used in the definition of a member function of a class X following the function's declarator-id or in the brace-or-equal-initializer of a non-static data member of class X shall be declared in one of the following ways:
  [...]
  - shall be a member of class X [...]

Summary:
  - argument types and trailing return types in member function declarations: only earlier-declared names are in scope, class is incomplete, name lookup does not find later-declared members.
  - argument types and trailing return types in member function definitions: all members are in scope, class is incomplete, name lookup does (p8) and does not (p7) find later-declared class members.
  - default arguments in member function declarations: all members are in scope, class is complete, name lookup does not find later-declared class members.
  - default arguments in member function definitions: all members are in scope, class is complete, name lookup does (p8) and does not (p7) find later-declared class members.
  - exception specifications in member declarations: only earlier-declared names are in scope, class is complete, name lookup does not find later-declared names.
  - exception specifications in member definitions: all members are in scope, class is complete, name lookup does (p8) and does not (p7) find later-declared names.
  - in-class member initializers: all members are in scope, class is complete, name lookup does (p8) and does not (p7) find later-declared names.

16. **(Richard Smith) disambiguating unparenthesized comma within in-class initializer**

Core issue#325 covers disambiguation of unparenthesized commas within default arguments, but the same issue affects brace-or-equal-initializers for non-static data members:

```
struct S {
  int n = T<a,b>(c); // ill-formed declarator for member 'b', or template?
};
```

clang behavior: As an extension of the proposed resolution to core issue#325, clang treats the first unparenthesized comma as the end of a brace-or-equal-initializer for a non-static data member.

priority: Low - this is the natural resolution, as an extension of core issue #325.

17. **(Sean Hunt) [class.ctor](12.1)/5, [class.copy](12.8)/11,23: missing arrays and variant members in a few cases for deleted defaulted member functions**

In the various specifications for when defaulted member functions get deleted, array types and variant members are occasionally forgotten. See the suggested resolution for a comprehensive list.

clang behavior: Somewhere between the suggestion and the standard as written (the code was written before the suggested resolution but with the intent of following it, and as a result not all cases added by the suggestion are covered).

suggested resolution: Change the first list in [class.ctor](12.1)/5 to read:

~~— 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-variant 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~~,~~ ;
— X is a union and all of 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)~~,~~ ;
— any direct or virtual base class, or non-static data member with no brace-or-equal-initializer, has class type M (or array thereof) and either M has no default constructor or overload resolution (13.3) as applied to M's default constructor results in an ambiguity or in a function that is deleted or inaccessible from the defaulted default constructor~~,~~ or, in the case of a variant member, is not trivial; or
— any direct or virtual base class or non-static data member ~~ha~~is of a class type (or array thereof) with a destructor that is deleted or inaccessible from the defaulted default constructor or, in the case of a variant member, is not trivial.

Similar changes should be made to the other forms of implicit member.

18. (Sebastian Redl) [expr.unary.noexcept](5.3.7) Destructor exceptions for temporaries in noexcept expressions.

The noexcept operator does not consider that the destruction of temporaries created in the operand expression could throw. Although throwing destructors ought to be rare, this can lead to bad surprises. Because throwing destructors ought to be rare, changing this should not lead to any issues with degraded performance or code breaking that shouldn't.

clang behavior: Clang follows the letter of the standard.

suggested resolution: Insert a new paragraph before the first that reads:

The operand of the noexcept operator is an unevaluated expression (Clause 5).

Change the first sentence of the first (now second) paragraph to read:

The noexcept operator determines whether the evaluation of its operand~~, which is an unevaluated operand,~~ as a full-expression can throw an exception (15.1).

Add a footnote to to this sentence that reads:

This means that calls to destructors of temporary objects created in the expression are considered.

priority: Moderate - implementations can easily be fixed, and diverging should not affect much code in the first place.

19. (Sean Hunt) [dcl.fct.def.default](8.4.2)/4: Functions other than special members cannot be user-provided.

The current definition of *user-provided* does not apply to any function other than a special member function. This has an unfortunate effect on aggregates, as a class with user-declared constructors will thus be an aggregate unless one of those constructors is a special member function.

clang behavior: clang currently implements the letter of the standard.

suggested resolution: Change (8.4.2) to read:
Explicitly-defaulted functions and implicitly-declared functions are collectively called *defaulted* functions, and the implementation shall provide implicit definitions for them (12.1 12.4, 12.8), which might mean defining them as deleted. A function is *user-provided* if it is user-declared and not explicitly defaulted or deleted on its first declaration.[*] A user-provided explicitly-defaulted function (i.e., explicitly defaulted after its first declaration) is defined at the point where it is explicitly defaulted; if such a function is implicitly defined as deleted, the program is ill-formed. [*Note:* Declaring a function as defaulted after its first declaration can provide efficient execution and concise definition while enabling a stable binary interface to an evolving code base. — *end note*]

Insert a footnote (at the point marked [*]) above reading:
Since only special member functions can be explicitly defaulted, any other function is user-provided unless deleted.

[Note that this resolution counter-intuitively allows a class with a deleted initializer list constructor to undergo aggregate initialization; it might be desirable to limit this in some fashion.]

priority: High - This is not a difficult issue to resolve, but may be cause for diverging implementations. Since this affects whether or not a structure is an aggregate, the behavior could be quite noticeable when list initialization is involved.

20. **(Sean Hunt) [except.spec](15.4)/14: Unspecified exception specifications of copy assignment operators with virtual bases.**

It's unspecified if an implicit copy assignment operator directly calls the copy assignment operators of virtual bases. Accordingly, this makes its exception specification unspecified, which is untenable. A change to 12.8 was made in 0x to deal with the issue of const parameter types, but this did not carry through to exception specifications.

Note that this issue is not new, but was previously unreported.

clang behavior: As suggested (including in C++03 mode).

suggested resolution: Insert the following sentence immediately before the note in [except.spec](15.4)/14: In the case of an implicit copy assignment operator for a class with virtual bases, assignment operators of virtual bases are considered directly invoked for this purpose as if they were direct bases, regardless of whether the definition of the special member function actually invokes them.

priority: Medium - it has gone unnoticed since C++98, but is now more significant with the addition of the noexcept operator. Divergence of implementations is quite possible, here, and could produce noticeably different code where templates are involved.

21. **(Richard Smith) [class.mem](9.2)/5: typedef and function members can have in-class initializers**

The grammar allows a brace-or-equal-initializer for typedef members and for member function declarations. There is no semantic rule making such uses ill-formed.

clang behavior: As suggested.

suggested resolution: Change [class.mem](9.2)/5 as follows:

A data member can be initialized using a *brace-or-equal-initializer*. (For static data members, see 9.4.2; for non-static data members, see 12.6.2). No other *member-declarator* shall have a *brace-or-equal-initializer*.

priority: Low.

22. **(Richard Smith) [stmt.ranged](6.5.4)/2: constexpr is permitted in for-range-declarations**

The resolution to CWG issue 1204 seemed to have only one effect: allowing constexpr in for-range-declarations. This change seems unrelated to the reported issue, and is in any case surprising since constexpr can (almost! see below) not appear in a legal for-range-declaration anyway.

Calling a constexpr member of a literal type does not appear to require the object expression in the member call to be a constant expression, so this example seems to be one (and possibly the only) legal way to use 'constexpr' in a for-range-declaration:

```
struct LiteralRange {
  constexpr LiteralRange() {}
  LiteralRange &begin();
```

```
    LiteralRange &end();
    bool operator!=(const LiteralRange &o) const;
    LiteralRange &operator++();
    constexpr int operator*() const { return 0; }
  };
  for (constexpr int a : LiteralRange()) {}
```

It is not clear whether the constexpr rules intended to allow constexpr member functions of non-constexpr objects of literal type to be called within constant expressions.

This change appears to be for consistency with the rules for conditions, but that consistency is not natural: a for-range-declaration is not like a condition and should not follow the same rules.

clang behavior: clang will reject constexpr within for-range-declarations.

suggested resolution: Either revert the changes made by CWG issue 1204 or change [stmt.ranged](6.5.4)/2 as follows:
In the decl-specifier-seq of a for-range-declaration, each decl-specifier shall be ~~either~~ a type-specifier ~~or constexpr~~.

priority: Low - it seems implausible that a real, legal program would ever use constexpr in a for-range-declaration.

23. (Richard Smith) [dcl.constexpr](7.1.5)/3,4,6: no restrictions on declarations of constexpr functions

(7.1.5)/6 says: 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.

While it's not completely clear which requirements are being referred to, the evolution of the wording indicates that it is the requirements in (7.1.5)/3, which only apply to function definitions, and some of them cannot be applied to function declarations. This results in the 'constexpr' status of some functions not being defined; for instance, the standard does not clearly specify whether the following type is literal:

```
template<typename T> struct S { constexpr S(T); };
struct T { T(); };
static_assert(std::is_literal_type(S<T>), "is S<T>::S(T) constexpr?");
```

The 'requirements for a constexpr function or constexpr constructor' fall into three categories: requirements on declarations (which can be checked after instantiating the declaration), syntactic requirements on definitions (which can be checked for template definitions prior to instantiation), and semantic requirements on definitions (which may require the template definition to be known at the point where the declaration is instantiated). The suggested resolution uses the first category for determining whether an instantiation is constexpr, and removes the third category entirely: such cases are already ill-formed (although no diagnostic is required), and it is neither reasonable to reject template instantiations on this basis, nor to cause them to mark the function as non-constexpr (since the definition is not necessarily known).

Also, the wording allows a constexpr constructor to be defined for a class with virtual base classes if the constructor is explicitly defaulted or deleted; this is a literal type:

```
struct B {}; struct D : virtual B { constexpr D() = delete; };
```

gcc behaviour: gcc appears to ignore (7.1.5)/6 when determining when an instantiation of a class template is a literal type: a non-aggregate template specialization class type seems to be considered literal if the class template had any constexpr constructors or ctor templates.

clang behavior: as per suggested resolution (WIP)

suggested resolution: See this separate paper for a suggested resolution for this and the other constexpr issues reported here.

priority: High - This renders constexpr unimplementable as specified. Implementations are already diverging: gcc always treats such instantiated declarations as constexpr, and clang will probably not do so.

24. (Richard Smith) [dcl.constexpr](7.1.5)/5,6 constexpr templates don't work

(7.1.5)/5 says "if no function argument values exist such that the function invocation substitution would produce a constant expression, the program is ill-formed; no diagnostic required".
(7.1.5)/6 says "If the instantiated template specialization of a constexpr function template [...] would fail to satisfy the requirements for a constexpr

function [...] that specialization is not a constexpr function"
The "requirements for a constexpr function" here are presumably those in (7.1.5)/3, and not (7.1.5)/5's nebulous ill-formed, NDR rule (since compiler vendors are not required to implement the (7.1.5)/5 checking, it seems infeasible for the constexpr-ness of a template instantiation to depend on it).

So what about a trivial case like this:

```
template<typename T> constexpr T max(T a, T b) { return (a > b) ? a : b; }
```

Suppose we instantiate this with a literal type T, with a non-constexpr operator>. (7.1.5)/6 implies that this is a constexpr function: it meets all the relevant requirements. But its result is unconditionally not a constant expression, so (7.1.5)/5 says this is ill-formed (NDR)! We should accept this case, instantiate a constexpr function, and merely treat the result as not a constant expression as per usual.

Conversely, if we consider (7.1.5)/5 to be part of the "requirements for a constexpr function", consider this:

```
template<typename T>
struct S { constexpr S(T a, T b) : c(a < b ? a : b) {} T c; };
```

Now we can observe whether (7.1.5)/5 checking is implemented in the compiler: if it is, the instantiation of S<T>::S<T>(T, T) will not be declared constexpr, the class has no non-copy, non-move constexpr constructors, and std::is_literal_type<S<T>> is derived from false_type.

clang behavior: I intend to implement the suggested resolution.

suggested resolution: Change [dcl.constexpr](7.1.5)/5 as follows:
For a constexpr function which is not an instantiation of a function template, if no function argument values exist such that the function invocation substitution would produce a constant expression (5.19), the program is ill-formed; no diagnostic required. For a constexpr function template definition, if no combination of template parameters and function argument values exist such that function template instantiation and function invocation substitution would produce a constant expression, the program is ill-formed; no diagnostic required. For a constexpr constructor which is not an instantiation of a function template, if no argument values exist such that after function invocation substitution, every constructor call and full-expression in the mem-initializers would be a constant expression (including conversions), the program is ill-formed; no diagnostic required. For a constexpr constructor template definition, if no combination of template parameters and argument values exist such that after function template instantiation and function invocation substitution, every constructor call and full-expression in the mem-initializers would be a constant expression (including conversions), the program is ill-formed; no diagnostic required. [Example: [...]]

priority: Low - implementations are not required to diagnose such issues, and do not currently do so.

25. **(David Majnemer) [over.literal](13.5.8)/8 User-defined literal example contains questionable literal suffix identifier**

The example given:

```
template <char...> int operator "" \u03C0(); // OK: UCN for lowercase pi
```

is supposedly a valid declaration of a literal operator template.

However, **[usrlit.suffix](17.6.4.3.5)/1** says: Literal suffix identifiers that do not start with an underscore are reserved for future standardization.

clang behavior: I intend to implement an extension warning that essentially says what **[usrlit.suffix]** has to say on the matter.

suggested resolution: The example should probably be changed to:

```
template <char...> double operator "" _\u03C0();
```

This example would not conflict with **[usrlit.suffix]**. Additionally, the return type of the literal operator template was adjusted to be a floating point type as it makes sense for a "pi operator". Alternatively, the restriction on a required leading underscore code be removed, making usage of user defined literals more natural.

priority: Very low - this is an issue in an example.

26. **(merged with issue 23)**

27. **(Sebastian Redl) [dcl.init.list](8.5.4)/3: List-initialization of references is very surprising.**

Consider the following code:

```
int i;
int &ri(i);
```

The reference ri is bound to i. The expectation for the user is that uniform initialization syntax works the same:

```
int i;
int &ri{i};
```

But this isn't the case. The FDIS has a list of 8 bullet points describing the meaning of initializers. The first one that is applicable to references is the fifth: "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 the temporary."
This bullet applies to the example above. So we list-initialize a temporary int from the list-initializer and try to bind the reference to that. Since we have a non-const lvalue reference, the binding fails.

If the reference is const, the situation is actually worse, because the code silently does something unexpected:

```
int i = 0;
```

```
const int &ri{i}; // binds to temporary
i = 4;
std::cout << ri << std::endl; // outputs "0"
```

DR934 attempted to resolve this, but failed to do so satisfactorily, and in any case the wording change from this issue, which was accepted into CD2, was lost in the FDIS. DR934 changed the wording of the fifth bullet to only apply to references to class type. This has the following effects:

```
int i = 0;
int &ri1 {i}; // ok: binds directly to i
const int &ri2 {i}; // ok: binds directly to i
const int &ri3 {}; // error: no longer binds to a value-initialized
                   // temporary, cannot value-initialize reference
```

Whether the third part here is a positive or negative change is a matter of taste. However, the situation for class types remains the same:

```
std::string s = "bar";
//std::string &rs1 {s}; // error: cannot bind to temporary
const std::string &rs2 (s); // ok: binds to s
const std::string &rs3 {s}; // ok: binds to temporary
s = "foo";
std::cout << rs2 << " " << rs3 << std::endl; // outputs "foo bar"
```

This is still the same surprising effect for users: an unexpected compile error for non-const references, and silent binding to a temporary for const references. Only now it is worse, because it is completely different behavior in class types and other types.
When a reference is initialized from a single-element initializer list, it should try to bind to the single element in all cases. The only case where this might not be what the user wants is when initializing a const reference to an array. We can special-case this if necessary, but the proposed solution for now is not to.

clang behavior: clang implements the proposed solution

suggested resolution: Swap the fifth and sixth bullet in [dcl.init.list]p3. This means that the bullet starting with "Otherwise, if the initializer list has a single element" comes before the bullet starting with "Otherwise, if T is a reference type".
Don't reintroduce the wording change of DR934, it only causes more confusion.

priority: High - divergent implementations are already coming into existence, and this is a significant change in semantics.

28. **(Richard Smith) [dcl.constexpr](7.1.5)/3,4 tag declarations only permitted in typedefs and using-declarations in constexpr functions**

The following is legal:

```
constexpr bool f() {
  typedef struct S dummy;
  return (S*)0;
}
```

The following is not:

```
constexpr bool f() {
  struct S;
  return (S*)0;
}
```

Since compilers are required to support the former, it is very strange to disallow the latter.

gcc behaviour: gcc accepts tag declarations and even definitions in constexpr functions, even in pedantic mode.

clang behavior: clang will allow both these cases (and possibly reject the latter with -pedantic)

suggested resolution: Add a new bullet to both (7.1.5)/3 and (7.1.5)/4 in the list of acceptable statements:
– simple-declarations that have empty init-declarator-lists and do not define classes or enumerations;

priority: Low - Implementations can simply accept this as an extension, and gcc already does.

29. **(Richard Smith) [dcl.constexpr](7.1.5)/4 constexpr union constructors are only legal for single-member unions**

A union constructor is only allowed to initialize a single non-static data member. A constexpr constructor is required to initialize all non-static data members. Therefore a constexpr union constructor can only exist in a union with a single non-static data member.

gcc behaviour: gcc does not check whether any members of a union are initialized in a constexpr union constructor.

suggested resolution: Change (7.1.5)/4 as follows:
– every non-static non-variant data member and base class sub-object shall be initialized (12.6.2);
– for a non-union class, one non-static data member shall be initialized in each non-empty anonymous union member;
– for a union, one non-static data member shall be initialized;

priority: Medium - implementations are diverging on what is legal, and do not implement what the standard says.

30. **(Richard Smith) [class.ctor](12.1)/6 When are defaulted default constructors constexpr?**

Defaulted default constructors are implicitly defined as constexpr if the "user-written default constructor would satisfy the requirements of a constexpr constructor".

There are two problems with this. Firstly, the rules only talk about the definition of a default constructor (see issue 23). Consider:

```
struct A {};
struct S1 : A {};
static_assert(!std::is_literal_type<S1>::value, "not literal yet");
constexpr S1 = S1(); // legal?
static_assert(std::is_literal_type<S1>::value, "now it's literal");
```

Since the implicit declaration of S1::S1 isn't constexpr, S1 has no constexpr constructors until S1::S1 is odr-used and implicitly defined.

Secondly, many reasonable programs are ill-formed under this rule, such as this one:

```
int f();
struct Bad { constexpr Bad(int n = f()) : n(n) {} int n; };
struct S2 { Bad b; } s;
```

This is ill-formed (no diagnostic required) because S2()'s implicit default constructor is defined constexpr but can never produce a constant expression. The following example is ill-formed (no diagnostic required) too, for the same reason:

```
constexpr int f(bool log) { return log ? (std::clog << '!', 0) : 0; }
struct Ugly { constexpr Ugly(bool log = true) : n(f(log)) {} int n; };
struct S3 { Ugly u; } u;
```

gcc behaviour: S1 is treated as a literal type. gcc does not diagnose that it has generated ill-formed default constructors for S2 and S3.

suggested resolution: Change (12.1)/6 as follows: If that user-written default constructor would satisfy the requirements of a constexpr constructor, the implicitly-defined default constructor is constexpr. An implicitly-declared default constructor is constexpr if the implicit definition would be.
Change (7.1.5)/5 as follows: For a user-declared constexpr constructor, if no argument values exist such that after function invocation substitution, every constructor call and full-expression in the mem-initializers would be a constant expression (including conversions), the program is ill-formed; no diagnostic required.

priority: Low - implementations seem to agree on how to treat this case.

31. **(Sean Hunt) [class](9)/6: How should template constructors affect trivially copyable-ness?**

The following class is trivially copyable:

```
  struct tc {
    template <typename T> tc(T&);
  };
```

The template constructor does not suppress triviality, but does participate in overload resolution and will be selected to perform the copy of any non-const object of that type. This has all sorts of nasty effects.

Tentative suggestion: Make templated constructors copy/move constructors and adjust other places to accommodate this change without affecting behavior of implicit members.

32. **(Richard Smith) [basic.types](3.9)/10: weird and pointless restriction on non-static data member initializers for literal types**

In the FDIS, the following extra restriction was added to the definition of a literal class type: "every constructor call and full-expression in the brace-or-equal-initializers for non-static data members (if any) is a constant expression (5.19),"

The bullet in question was added to resolve core issue 1219, which was actually not a defect: since a class type with a brace-or-equal-initializer is never an aggregate, the class is already required to have a constexpr constructor (by the very next bullet). Either the (defaulted or user-provided) constexpr constructor initializes the member, in which case the brace-or-equal-initializer should be ignored, or it does not, in which case the program is already ill-formed by [dcl.constexpr]/4 in the appropriate circumstances. Hence this rule only rejects reasonable programs.

Further, the rule rejects reasonable programs which have constexpr constructors which use the initializers, such as this one:

```
struct S {
  int a = 0, b = a;
  constexpr S(int k) : a(k) {} };
 ... since b's initializer is not a constant expression.
```

Even the rules in [dcl.constexpr]/4 are too strong; they too rule out the above program. The bullet on brace-or-equal-initializers there is unnecessary, since if used, they are treated as member initialization ([class.base.init](12.6.2)/8) and thus are covered by the normal rules on initializing non-static data members.

clang behavior: clang will probably implement the suggested resolution.

suggested resolution: Delete that bullet entirely. Also delete the following bullet in [dcl.constexpr](7.1.5)/4: "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"

priority: Low-Medium - there is a risk of implementations diverging.

33. (Richard Smith) [dcl.constexpr](7.1.5)/6 constexpr functions inside non-class templates

This paragraph says "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 [...]"
This does not cover all cases where a function could be defined inside a template:

```
template<typename T> void f() {
  struct S {
    constexpr T g() { return T(); }
  };
  S().g();
}
```

Here, f()::S::g is neither a function template nor a member function of a class template.
priority: Low - this is a special case of a more general, old problem.

34. (Richard Smith) [basic.def.odr](3.2)/4 must T be complete in an implicit conversion to T&?

The sixth bullet in the note in this paragraph says "A class type T must be complete if [...] an expression [...] is converted to type pointer to T or reference to T using an implicit conversion". There appears to be no normative wording to justify this in cases such as:

```
struct A;
struct C { operator A&(); } c;
A &a = c;
```

priority: Low - this text is not normative.

35. (Richard Smith) [class.inhctor](12.9)/2 inherited constructors are constexpr too often

According to the FDIS, C is a literal type:

```
struct A {
  constexpr A();
};
struct B {
  B();
};
struct C : A {
  B b;
  using A::A;
}; ... because constexpr is a constructor characteristic, and thus C::C is
```

constexpr irrespective of whether it obeys the relevant restrictions.
Further, the above code is ill-formed (no diagnostic required) because C::C() is constexpr but cannot produce a constant expression (however, this additional problem is covered by the solution to issue 30 above).
priority: Low-Medium - there is a risk of implementations diverging here.

36. (Sean Hunt) [class](9)/6: A class with multiple default constructors can be needlessly trivial.

A class is trivial if it has a trivial default constructor and is trivially copyable. This does not cover the case where a class has multiple default constructors, and one of them is trivial, as in the following:

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

As a result, A is considered trivial. It probably should not be because any default-initialization will be ill-formed, but the triviality may affect other things (such as POD-ness and thus whether initialization is required).

clang behavior: As the suggested resolution.

suggested resolution: Replace “has a trivial default constructor” with “has a unique and trivial default constructor”.

priority: Low - default arguments on constructors to create ambiguity like this are uncommon enough, and the odds are good that they wouldn't stay that way for long due to other issues. It's not entirely clear anything bad would happen if implementations handled this differently (since, in practice, the worst that will happen is some undefined behavior that does the right thing anyway).