Document number: J16/03-0126 = WG21 N1543
Date: 14 November, 2003
Reply to: William M. Miller
The MathWorks, Inc.
wmm@world.std.com

Analysis and Proposed Resolution for Core Issue 39

0. Introduction

Issue 39 has been one of the most intractable and frustrating problems the Core Language Working Group has dealt with. After four years of deliberation, a resolution was approved in October, 2002, and was moved to "Ready" status in April, 2003.

Unfortunately, a discussion in the `comp.std.c++` newsgroup[1] in July, 2003, led me to the conclusion that the proposed resolution is flawed. I posted a message at that time about my concerns to the core group email reflector[2], but there was no comment from other members of the group. More recently, I elaborated on that analysis with a proposed alternative resolution[3], but again no one replied.

This paper is the result of combining two documents I produced in preparation for the Kona meeting with the result of the core group deliberations, with some minor changes and elaborations (discussed in footnotes).

The next section contains a summary view of what lookup in class scope is attempting to achieve, while succeeding sections provide analysis of the current wording of the Standard, the fundamental defect that is reported in issue 39, and the pre-Kona proposed resolution. Those who already feel comfortable with this background information can skim it or skip directly to Section 5 and following. These sections cover the reasons I think the pre-Kona proposal is flawed, an unneeded restriction in both the current wording and the earlier proposed resolution, and a revised proposal reflecting the core group discussions in Kona.

1. Concepts of Class-Scope Lookup

The purpose of name lookup is to associate a given use of a name with one or more declarations of that name (3.4¶1). The fundamental concepts involved in looking up a name in class scope are *inheritance*, *hiding*, *dominance*, and *ambiguity*.

Inheritance and *hiding* are the two most important influences on class-scope lookup. A member of a base class is also a member of a class derived from it, so names declared in base classes are normally visible in derived classes as well. A name declared in a derived class, however, *hides* any declarations of the same name in its base class(es). Consequently, looking up that name in the derived class scope will find the derived class declaration(s) and not any that are inherited.

Dominance is a variation on ordinary name hiding. There may be multiple paths through the

1 See `http://tinyurl.com/q8zn` to read the thread via Google Groups. Note in particular the exchange between Daveed Vandevoorde and me for the portion of the discussion most directly relevant to this issue.
2 `http://www.research.att.com/~ark/cgi-bin/wg21/message?wg=core&msg=10033`
3 `http://www.research.att.com/~ark/cgi-bin/wg21/message?wg=core&msg=10165`

inheritance DAG from a derived class to a given base class, and a name declared in that base class might be visible on some paths and hidden on others. If the base class in question is a virtual base and lookup finds both a declaration from that class and one that hides it, the hiding declaration is chosen. For example,

```
struct B {
  int i;
};

struct I: virtual B {
  int i;
};

struct D: I, virtual B {
  void f() {
    i = 2;      // I::i -- B::i is dominated by I::i
  }
};
```

Example 1

There are two different kinds of *ambiguity* that can result from a class member reference or *qualified-id*, and it is important to distinguish between them. I will call the first kind *multiple-declaration ambiguity*. It results when lookup finds more than one declaration of a name and the names do not form an overload set:

```
struct B1 {
  int i;
  void f();
};

struct B2 {
  int i;
  void f(int);
};

struct D: B1, B2 {
  void g() {
    i = 5;      // ambiguous: B1::i or B2::i?
    f(0);       // ambiguous: B1::f() or B2::f(int)?
  }
};
```

Example 2

Note that ambiguity detection precedes and is distinct from overload resolution: the presence of the argument in the attempt to call `B2::f(int)` does not disambiguate the reference to the

name `f`. Because `B1::f()` and `B2::f(int)` are declared in different scopes, they do not form an overload set, causing any use of the name `f` to be ambiguous.

The second kind of lookup ambiguity is what I call *multiple-subobject ambiguity*. This kind of ambiguity occurs when the name lookup finds the declaration of a nonstatic data member or nonstatic member function in a base class `B` and there is more than one `B` base class subobject in the class in which the lookup occurs. For example,

```
struct B {
   int i;
   void f();
};

struct I1: B { };
struct I2: B { };

struct D: I1, I2 {
   void g() {
      i = 5;      // ambiguous: I1's B::i or I2's B::i?
      f();        // ambiguous: I1's B::f() or I2's B::f()?
   }
};
```

Example 3

It's important to note that this has historically been a *lookup* ambiguity, dating back to the ARM and continuing through the pre-Kona proposed resolution for issue 39. In these formulations, it has nothing to do with, for instance, the conversion of the `this` pointer from `D*` to `B*` in the call to `B::f()`. The ambiguity is based solely on the fact that the lookup found a nonstatic declaration and that there are multiple base class subobjects of the class containing that declaration.

2. Exploring the Current Wording of the Standard

The current description of class-scope lookup is found in 10.2¶2:

> The following steps define the result of name lookup in a class scope, `C`. First, every declaration for the name in the class and in each of its base class sub-objects is considered. A member name `f` in one sub-object `B` *hides* a member name `f` in a sub-object `A` if `A` is a base class sub-object of `B`. Any declarations that are so hidden are eliminated from consideration. 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*.[96] 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. Otherwise that set is the result of the lookup.
>
> [96] Note that *using-declaration*s cannot be used to resolve inherited member ambiguities; see 7.3.3.

The first thing to note from this wording is that lookup is described completely in terms of base class subobjects, rather than base/derived class relationships. This means, for instance, that in example 3 above, the lookup set for `i` contains *two* declarations – both `I1`'s and `I2`'s `B::i` – from the outset ("... every declaration for the name ... in each of its base class sub-objects..."). This is a very different perspective from that of the rest of the Standard, where member declarations are typically considered to belong to classes rather than objects, and the shift is both significant and easily overlooked.

The second important point of interest is how hiding is handled. Because the description is in terms of subobjects, dominance is an implicit outcome rather than being described explicitly. To see how this works, consider **Example 1** above. Because `B` is a base class subobject of `I`, `I::i` hides `B::i`. However, there is only a single `B` base class subobject in `D` and consequently only a single declaration of `B::i` in the original lookup set. Removing it because of the hiding declaration in `I` means that it cannot be found by the lookup, even though `D` directly inherits it from `B`.

The next notable feature in the current wording is the implementation of multiple-declaration ambiguity detection: "If the resulting set of declarations are not all from sub-objects of the same type... there is an ambiguity." Translating this to the more usual class-based perspective, it means that, after consideration of inheritance and hiding, all the remaining declarations must be direct (not inherited) members of the same class.

There are two practical implications of this requirement. The first is that it prevents the result of the lookup from containing incompatible declarations. According to 3.4¶1, the result of a lookup can contain more than one declaration only if the the declarations form a set of overloaded functions. Requiring that all the declarations found by a lookup be members of a single class enforces this restriction because a heterogeneous set (e.g., one containing both a data member and a function) cannot be declared in a single scope (3.3¶4). The only way such a set could result is via inheritance from two different classes, and this rule precludes that possibility.

The other consequence of note is that an overload set must be created explicitly by declarations in a single scope; it cannot be the implicit result of inheritance, as noted above in the discussion of **Example 2**. The rationale for this restriction is that it prevents hijacking of overload

resolution. Consider the following example:

```
struct B {
  void f(int);
};

struct libA { };

struct D: B, libA { };

void g(D* dp, short s) {
  dp->f(s);      // calls B::f(int)
}
```

Example 4

Assume that class `libA` comes from an externally-supplied library. If a new release of this library added the function `libA::f(short)`, the call `dp->f(s)` would be quietly redirected from `B::f(int)` to the new function, were it not for the prohibition of creating implicit overload sets.

The next feature to note from the current wording is the implementation of multiple-subobject ambiguity detection: "If ... the set has a nonstatic member and includes members from distinct sub-objects, there is an ambiguity." As discussed above in the commentary on **Example 3**, this is a lookup ambiguity and applies regardless of whether the declaration(s) are data members or member functions and without consideration of the context in which the name reference occurs – function call, forming a pointer-to-member, etc. Furthermore, it applies to mixed sets of static and nonstatic member functions: even if overload resolution would have picked a static member function from the set, multiple-subobject ambiguity is diagnosed if any of the member functions are nonstatic.

The final item of interest from the Standard wording is the treatment of *using-declaration*s: "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*." I call this approach *transparent using-declarations*. Naturally, the other technique (in which a *using-declaration* is considered to belong to the class in which it appears) will be termed *opaque using-declaration*s.

The principal reason for choosing transparent *using-declaration*s is, I believe (with support from the attached footnote and the more complete discussion in 7.3.3¶14), to detect multiple-subobject ambiguities. Consider, for instance,

```
struct B {
  void f();
};

struct I1: B { };
struct I2: B { };

struct D: I1, I2 {
  using B::f;
  void g() {
    f();      // ambiguous: multiple B subobjects
  }
}
```

Example 5

If the `using B::f;` declaration were treated as opaque, the lookup of `D::f` would be unambiguous, even though there is clearly an ambiguity – there are two `B` subobjects, and the function expects a pointer to one of them as its implicit `this` parameter. The Standard wording handles this situation by treating the `using B::f;` declaration as coming from each `B` subobject. Consequently, the lookup set contains two declarations of `B::f` and, because `B::f` is nonstatic, a multiple-subobject ambiguity is detected.

Treating *using-declaration*s as transparent also prevents the spurious detection of ambiguities that aren't really there:

```
struct B {
  void f();
};

struct I1: virtual B {
  using B::f;
};

struct I2: virtual B {
  using B::f;
};

struct D: I1, I2 {
  void g() {
    f();      // unambiguous
  }
};
```

Example 6

Opaque *using-declaration*s would lead to diagnosis of a multiple-declaration ambiguity in this

case where none actually exists. Because the *using-declaration*s are transparent, and because there is only a single `B` subobject in `D`, there is only a single declaration in the lookup set and the reference is unambiguous.

3. Problems with the Standard's Wording

The first problem is the one that gave rise to issue 39 and is illustrated by the following example:

```
struct A {
  int x(int);
};

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

int f(B* b) {
  b->x(3);       // ambiguous
}
```

Example 7

Under the current wording of the Standard, this is a multiple-declaration ambiguity. Because *using-declaration*s are treated as transparent, the lookup set for `x` in `B` contains both `A::x(int)` and `B::x(float)`. These are obviously not both "from sub-objects of the same type," so a multiple-declaration ambiguity exists.

Equally obviously, this outcome is not what was intended. The example in 7.3.3¶12 is essentially identical to **Example 7**, and the wording in 7.3.3¶13 clearly assumes that lookup will succeed in such cases and allow overload resolution to proceed:

> For the purpose of overload resolution, the functions which are introduced by a *using-declaration* into a derived class will be 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.

A similarly problematic case (not described in issue 39) is suggested by **Example 4** above. As noted in the commentary there, the restriction that all declarations must come from a single class exists expressly to prevent implicitly combining base class scopes into an overload set. If one wanted explicitly to create a merged overload set, the obvious way to do so would be via *using-declaration*s:

```
struct fixed {
   int f(int);
   long f(long);
};

struct float {
   float f(float);
   double f(double);
};

struct arith: fixed, float {
   using fixed::f;
   using float::f;
};
```

Example 8

Again because of transparent *using-declaration*s, any attempt to look up `f` in the scope of `arith` will be *both* a multiple-declaration ambiguity (not all declared in the same class) *and* a multiple-subobject ambiguity ("...has a non-static member and includes members from distinct sub-objects," namely `fixed` and `float`).

A separate problem with the current wording, not directly related to *using-declaration*s, was reported in issue 306: what happens when a type name is found in two different scopes? The following example is one illustration of the issue:

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

struct C : public A, public A::B {
   B *p;
};
```

Example 9

The lookup for `B` in the scope of `C` finds two different declarations, one in each base class. In `A`, the declaration that is found is the class `B` itself. In `A::B`, it is the injected-class-name (9¶2, 3.4¶3). The fact that both these declarations name the same type is irrelevant; because they are not "from sub-objects of the same type," there is a multiple-declaration ambiguity. As the text of the issue points out, `typedef`s can also pose the same problem.

Another problem with the current wording is not yet on the issues list; I noticed it while

investigating issue 39. According to 3.3.7¶2,

> A class name (9.1) or enumeration name (7.2) can be hidden by the name of an object, function, or enumerator declared in the same scope. If a class or enumeration name and an object, function, or enumerator are declared in the same scope (in any order) with the same name, the class or enumeration name is hidden wherever the object, function, or enumerator name is visible.

There is nothing in this description to indicate that class scope is exempted from this behavior – it simply says "the same scope," without exception – but the description of hiding in 10.2¶2 omits any mention of the "struct stat" (or "scope-and-a-half") hack. Readers of the Standard tend to approach 10.2 as a complete, self-contained description of looking up a name in class scope, and there has been some confusion as to whether the "struct stat" hack applies.

The intent of the Committee was that the "struct stat" hack should apply to class scope as it does to all other scopes (cf core issue 400). The purpose of the hack is to enhance compatibility with C programs, where struct and enumeration tags are in a separate "namespace" and thus cannot conflict with object, function, and enumerator names. Even though C does not have class scope, it does allow lexical nesting of struct definitions. Thus, exempting class scope from the "struct hack" would result in incompatibility with C in cases like the following:

```
struct S {
   struct X { /* ... */ };
   int X;
};
```

Example 10

I did not do a thorough survey of implementations, but the EDG compiler, at least, agrees with this interpretation and applies the "struct stat" hack to class scope.

It could be argued with some justification that the specification in 3.3.7¶2 is intended to be implicitly understood in the 10.2¶2 description. However, the fact that base/derived hiding is mentioned in 3.3.7¶1,3 and yet described again in detail in 10.2¶2 could lead readers to conclude that the latter passage is intended to be a complete *de novo* specification of all aspects of lookup in class scope. Indeed, informal discussions have indicated some level of confusion about the question of whether the hack applies to class scope. I believe that the intent should be made explicit in 10.2.

Finally, a similar confusion has arisen regarding whether the "transparency" of object declarations when looking up the name in an *elaborated-type-specifier* is intended to apply to

class scope.[4] Consider the following example:

```
struct B {
  struct X { };
};

struct I: B {
  void X();
};

struct D: I { };

struct D::X x;      // Ill-formed?
```

Example 11

Because the current version is worded in terms of "removing declarations from consideration," it could be argued that the normal processing described by 3.4.4 cannot apply to class scope lookup. Again, I believe this is an indication that 10.2 should be a complete specification of how to look up a name in class scope.

4. The Pre-Kona Proposed Resolution of Issue 39

At the April, 2003, meeting in Oxford, the Core Working Group agreed that the proposed resolution for issue 39 correctly addressed the defect and moved it to "Ready" status, in preparation for a vote by the full Committee at the October, 2003, meeting to accept it as an official Defect Report. This resolution takes a completely different approach from that currently found in the Standard and would replace 10.2¶2 with the following:

> 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-declaration*s) were found. In the declaration set, *using-declaration*s 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

4 See http://www.research.att.com/~ark/cgi-bin/wg21/message?wg=core&msg=10198.

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 subobject 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-declaration*s 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*]

The specification in the Standard is based on essentially the following steps:

- Create a set of declarations of the name, with one for each subobject in the DAG whose class directly contains such a declaration.
- Remove any hidden declarations.
- Test the results for violation of the ambiguity rules.

In contrast, the proposed resolution defines a recursive algorithm for traversing the inheritance tree (not a DAG, because the algorithm does not distinguish between virtual and non-virtual bases in the traversal) and accumulating the results of the lookup. This algorithm has the following characteristics:

- The traversal stops when it encounters a class containing a declaration of the name being looked up, including a *using-declaration* – that is, it does not examine any base classes of a class that contains such a declaration. This feature provides the implementation of ordinary base/derived hiding. It is also the source of opacity for *using-declaration*s.
- The first step of the merge phase of the algorithm ("If each of the sub-object members of [one set] is a base class subobject of at least one of the subobject members of [the other set]...") is the implementation of dominance. A virtual base class may be visited many times, but a set in which the virtual base appears directly will be trumped by a set in which it is a base-class subobject, i.e., one that contains a dominating declaration.
- The second step of the algorithm ("if the declaration sets differ") is the check for multiple-declaration ambiguity. Even if a multiple-declaration ambiguity is detected, however, the subobject sets are maintained to allow the dominance check to proceed in more-derived classes. It is here that the transparency of *using-declaration*s ("*using-declaration*s are replaced by the members they designate") comes into play,

suppressing the multiple-declaration ambiguity when separate *using-declaration*s designate the same entity.

- The third step screens for multiple-subobject ambiguity: if any member of the declaration set is a nonstatic member, the subobject sets being merged must have only a single subobject of the class containing that member. Again, the comparison takes advantage of transparent *using-declaration*s.

- Issue 306 is dealt with by the statement that "type declarations (including injected-class-names) are replaced by the types they designate."

It's instructive to apply this algorithm to the sample code from issue 39 (**Example 7** above). Because `B` contains a declaration of the name being looked up, `x`, the algorithm's terminal condition is immediately satisfied: "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." Thus, after replacing the *using-declaration* by the member it designates, the result of the lookup is `{A::x(int), B::x(float)}` and there is no ambiguity, which is the desired result.

5. Problems with the Pre-Kona Proposed Resolution

I believe there are three flaws in the pre-Kona version of the proposed resolution of issue 39. Although only the first of these is serious, I'll discuss the other two briefly at the end of this section.

To begin the discussion of the major problem with the algorithm, it may be helpful to consider the example in 7.3.3¶14, contrasting how the Standard wording and the proposed resolution handle it. (The question posed by the original poster in the `comp.std.c++` thread mentioned earlier was, "Why is this example ambiguous?") For ease of reference, the example is as follows:

```
struct A { int x(); };

struct B : A { };

struct C : A {
   using A::x;
   int x(int);
};

struct D : B, C {
   using C::x;
   int x(double);
```

```
};

int f(D* d) {
   return d->x(); // ambiguous: B::x or C::x
}
```

Example 12

The analysis according to the Standard wording runs as follows. First, create a set containing every declaration of `x` in `D` and each of its subobjects, considering declarations introduced by *using-declaration*s as coming from each subobject of the type containing the declaration designated by the *using-declaration*. `D` has a `B` subobject, a `C` subobject, and two `A` subobjects. This results in the following set of declarations:

```
{ A::x() in B,
  A::x() in C,
  C::x(int),
  D::x(double) }
```

This set fails the multiple-subobject ambiguity test: "If ... the set has a nonstatic member and includes members from distinct subobjects, there is an ambiguity."

The application of the proposed resolution to this example is essentially identical to the analysis of **Example 7** given at the end of the preceding section. Because `D` contains a declaration of `x`, the algorithm's terminal condition is immediately satisfied. After replacing *using-declaration*s by the members they designate, the resulting lookup set S(x,D) is

```
{ { A::x(),
    C::x(int),
    D::x(double) },
  { D } }
```

According to the proposed resolution, "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." In this case, the declaration set is

```
{ A::x(),
  C::x(int),
  D::x(double) }
```

It is not invalid, therefore *there is no lookup ambiguity*. Instead, overload resolution occurs, selecting `A::x()`, the call expression is processed (5.2.2¶1,4), and there is a *conversion* ambiguity attempting to convert `d` to type `A*` in order to pass it to the implicit `this` parameter of `A::x()` (4.10¶3). In other words, under the proposed resolution, the example is still

diagnosed as ambiguous, but it is a different *kind* of ambiguity.

Unfortunately, this safety net only works for member functions, not for data members. Consider the following example:

```
struct B {
  int i;
};

struct I1: B { };
struct I2: B { };

struct D: I1, I2 {
  using B::i;
};

int f(D* dp) {
  return dp->i;
}
```

Example 13

Here the result of the lookup is `B::i`, a valid declaration set, and there is nothing elsewhere in the Standard to diagnose this as an ambiguity.[5]

The problem illustrated by these examples is that the algorithm in the proposed resolution only detects ambiguities while merging base class lookup sets into the result set of a derived class. Because there is no lookup in the base classes of a class containing a declaration of the name being looked up, there is no merge step for such a class, and ambiguities that should have been diagnosed are overlooked.

As noted above, the remaining flaws in the proposed resolution are minor and can be very easily repaired. The second shortcoming is one that the proposed resolution shares with the Standard's wording: as noted above, the "struct stat" hack is not explicitly described or incorporated, nor is the special processing for lookup of names in *elaborated-type-specifier*s, *base-specifier*s, etc.

The third flaw in the proposed resolution is that it doesn't appear to handle empty lookup sets. Consider the first bullet in the description of the merge processing:

5 During the core group discussions in Kona, it was thought that the resolution of issue 52 in 11.2¶5 addressed this need. However, a closer reading of that paragraph shows that the object expression must be convertible to the *naming class* of the member reference, not to the class of the member, so it still appears that there is nothing in the Standard outside the lookup rules to make such data member references ambiguous.

> 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).

The case where there are no subobject members of one set or the other seems not to be handled by either of these conditions, falling through into the next bullet:

> Otherwise, if the declaration sets of S(f,B_i) and s(f,C) differ, the merge is ambiguous.

That's clearly not intended, because every lookup in which a declaration of the name is found only in a base class of the one being searched would result in an ambiguity. I think this was probably intended to be read as something like

> 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), **or if S(f,B_i) is empty,** 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), **or if S(f,C) is empty,** the new S(f,C) is a copy of S(f,B_i).[6]

6. Reducing the Scope of Multiple-Subobject Ambiguity Detection

As previously noted, the current wording is very broad in its application of the test for multiple-subobject ambiguity: if *any* member of the lookup set is nonstatic, none of the declarations are allowed to be from multiple subobjects. Furthermore, the context of the name reference is irrelevant – whether it's a function call or forming a pointer-to-member, whether overload resolution might choose a static member function, etc., are all ignored in the detection of multiple-subobject ambiguities. The pre-Kona proposed resolution maintains this approach.

I believe this policy is unnecessarily restrictive and that the Committee should take advantage of this defect resolution to consider relaxing it. There are at least three kinds of situations in which this ambiguity is needlessly diagnosed. First, consider code like the following:

```
struct B {
  void f();
```

6 Jason Merrill confirms that this was the intention, and that the "each of..." clauses were meant to subsume the "empty" case. However, the English phrase "each of" typically presumes the existence of at least one of the items being referred to, so it seems better to give an explicit treatment of the "empty" cases.

```
  static void f(int);
};

struct I1: B { };
struct I2: B { };

struct D: I1, I2 {
  void g() {
    f(0);       // ambiguous
  }
};
```

Example 14

In this example, an overload set that comes from multiple subobjects contains both a static and a nonstatic member function. If the ambiguity were not diagnosed during name lookup, overload resolution would unambiguously choose the static function. Also, as noted above in the discussion of **Example 12**, even if overload resolution chose a nonstatic member function, the ambiguity would still be diagnosed because of the attempt to convert the object expression to the type of the implicit `this` parameter.

A different kind of situation that leads to the same result is illustrated by the following:

```
struct B1 {
  void f();
};

struct B2 {
  void f(int);
};

struct I1: B1 { };
struct I2: B1 { };

struct D: I1, I2, B2 {
  using B1::f;
  using B2::f;
  void g() {
    f(0);       // ambiguous
  }
};
```

Example 15

Here, the desired function is a member of a single base class subobject; it is only the fact that the other member of the overload set comes from multiple subobjects that causes it to be diagnosed

as ambiguous. Again, the safety net applies: if overload resolution picked the function from the multiple subobject, the conversion ambiguity would still be detected.

The third example is perhaps the most persuasive:

```
struct B {
  void f();
};

struct I1: B { };
struct I2: B { };

struct D: I1, I2 { };

void (B::* pmf)() = &D::f;      // ambiguous
```

Example 16

If it weren't for the expansive wording of the multiple-subobject ambiguity rule, there would be no reason to treat this as ambiguous at all. The type of the expression `&D::f` is `void (B::*)()` – i.e., nothing at all reflecting the existence of the base class subobjects. It is a pointer-to-member that could be used with any "`B`" object expression, not tied to `D` in any way. (Attempting to convert it to `void (D::*)()` would, of course, be ambiguous.)

A name that resolves to a nonstatic member function can only be used to call the function or to form a pointer-to-member (5.1¶10). In a call expression, multiple-subobject ambiguity will be detected by the attempt to convert the object expression to the implicit `this` parameter. When forming a pointer-to-member, there is no intrinsic multiple-subobject ambiguity to diagnose; the only multiple-subobject ambiguity that can arise is in the context of a conversion, and that is handled in 4.11¶2.

Also, the usual practice in C++ is only to diagnose erroneous function declarations and uses if they are selected by overload resolution (cf 7.3.3¶11, indistinguishable functions introduced by *using-declaration*s; 11¶4, access control; 13.3.3¶4, conflicting default arguments).

In light of all these considerations, the only rationale for diagnosing multiple-subobject ambiguity among member functions as a *lookup* ambiguity would seem to be the parallel with nonstatic data members. As noted above, multiple-subobject ambiguity among nonstatic data members is generally only diagnosed in the description of name lookup – but even for data members, there is no reason that forming a pointer to data member should be subject to multiple-subobject ambiguity checking. (The analysis in **Example 16** applies equally well to pointers to data members.)

7. New Proposed Resolution

Discussions among the core group in Kona affirmed the idea of addressing the issues raised in both item 5 and item 6 in a single revised proposed resolution. As previously noted, treating both multiple-declaration and multiple-subobject ambiguity as lookup ambiguities dates back to the ARM. However, there is no fundamental requirement that they be handled together. Removing multiple-subobject ambiguity from the description of class-scope lookup and leaving it to be dealt with in each applicable expression context would obviate the major problem with the pre-Kona resolution described in part 5, simplify the merge algorithm by removing the check for multiple-subobject ambiguity, and make the problematic examples in part 6 well-formed.

The following proposed wording is based on this approach, as well as incorporating changes to address the other issues described in part 5.

1) Change 10.2¶2 to read:

> 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-declaration*s) were found. In the declaration set, *using-declaration*s 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 declared in C that satisfies the requirements of the language construct in which the lookup occurs. [*Note:* Looking up a name in an *elaborated-type-specifier* (3.4.4) or *base-specifier* (clause 10), for instance, ignores all non-type declarations, while looking up a name in a *nested-name-specifier* (3.4.3) ignores function, object, and enumerator declarations. As another example, looking up a name in a *using-declaration* (7.3.3) includes the declaration of a class or enumeration that would ordinarily be hidden by another declaration of that name in the same scope.] If the resulting declaration set is not empty, the subobject set contains C itself, and calculation is complete.
>
> Otherwise (i.e., `C` does not contain a declaration of `f` or the resulting declaration set is empty), S(f,C) is initially empty. If C has base classes, calculate the lookup

set for f in each direct base class subobject 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), or if S(f,B_i) is empty, 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), or if S(f,C) is empty, 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, 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*]

2) Turn the non-example text of 10.2¶4-6 into notes.[7]

3) Add the following text as a new paragraph following the current 10.2¶7:

> [*Note:* Even if the result of name lookup is unambiguous, use of a name found in multiple subobjects might still be ambiguous (4.11, 5.2.5, 11.2).] [*Example:*
>
> ```
> struct B1 {
>   void f();
>   static void f(int);
>   int i;
> };
>
> struct B2 {
>   void f(double);
> };
>
> struct I1: B1 { };
> struct I2: B1 { };
>
> struct D: I1, I2, B2 {
>   using B1::f;
>   using B2::f;
>
>   void g() {
>     f();                        // Ambiguous conversion of this
>     f(0);                       // Unambiguous (static)
>     f(0.0);                     // Unambiguous (only one B2)
>     int B1::* mpB1 = &D::i;     // Unambiguous
>     int D::* mpD = &D::i;       // Ambiguous conversion
>   }
> };
> ```
>
> —*end example*]

4) Add the following text as a new paragraph following 5.2.5¶4:[8]

> 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.

7 The pre-Kona proposed resolution included converting paragraphs 5 and 6 into notes. The core group did not discuss paragraph 4, but I believe it is equally non-normative.

8 As previously noted, the core group thought that 11.2¶5 dealt adequately with data member multiple-subobject ambiguity, but that is not the case. Implementing the approach of removing multiple-subobject ambiguity thus requires an explicit treatment, and this seemed to me to be the best place.