Document Number: J16/04-0066 = WG21 N1626
Date: 2004-04-04
Reply To: William M. Miller
The MathWorks, Inc.
`wmm@world.std.com`

Proposed Resolution for Core Issue 39 (Rev. 1)

I. Introduction

The previous version of this paper (J16/03-0126 = WG21 N1543) contains an in-depth analysis of Core Language issue 39 and proposed Working Paper wording changes to implement the deliberations of the Core Language Working Group at the Kona (October, 2003) meeting. That wording was discussed at the Sydney, Australia (March, 2004) meeting, resulting in some small modifications. This paper contains the complete wording of the proposed resolution, for ease of reference.

The wording herein is the same as in the earlier paper, with two exceptions:

- The revision proposed for 5.2.5¶4 was modified as described in message c++std-core-10295, and the wording was changed slightly as a result of the CWG's review.
- In response to the concerns raised in message c++std-core-10442, the proposal now contains a revision to the text of 5.2.2¶4.

II. Detailed Wording Changes

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 east 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.

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:

> If `E2` is a non-static data member or a non-static member function, the program is ill-formed if the class of which `E2` is directly a member is an ambiguous base (10.2) of the naming class (11.2) of `E2`.

5) Change 5.2.2¶4 as follows:

> If the function is a nonstatic member function, the "`this`" parameter of the function (9.3.2) shall be initialized with a pointer to the object of the call, converted as if by an explicit type conversion (5.4). [*Note:* There is no access **or ambiguity** checking on this conversion; the access checking **and disambiguation are** ~~is~~ done as part of the (possibly implicit) class member access operator. See **10.2,** 11.2**, and 5.2.5**.]