Doc. No.: X3J16/96-0213
WG21/N1031
Date: November 13, 1996
Project: Programming Language C++

Reply to: Josee Lajoie
josee.vnet.ibm.com

CORE WG 1 - KONA MOTIONS

1) Motion (to resolve issue 666 and clarify that a class name used in an elaborated-type-specifier or as a base class name is not hidden by a namespace name):

Move we:

-- replace the first sentence of 3.4.4 paragraph 1 with:

An elaborated-type-specifier may be used to refer to a previously-declared class-name or enum-name even though the name is hidden by a non-type declaration (3.3.7).

-- replace the third sentence of 10 paragraph 1 with:

During the look up for a base class name, non-type names are ignored (3.3.7).

2) Motion (to resolve issue 727 and describe that block extern declarations and and function block declarations refer to members of the immediately enclosing namespace only):

Move we:

-- replace the text of 3.5 paragraph 6 before the example with:

The name of a function declared in block scope, and the name of an object declared by a block scope extern declaration, have linkage. If there is a visible declaration of an entity with linkage having the same name and type, ignoring entities declared outside the innermost enclosing namespace scope, the block scope declaration declares that same entity and receives the linkage of the previous declaration. If there is more than one such matching entity, the program is ill-formed.

-- move 7.3.1.2 paragraph 4,including example, immediately following 3.5 paragraph 6 and replace the first two sentences with:

When a block scope declaration of an entity with linkage is not found to refer to some other declaration, then that entity is a member of the innermost enclosing namespace.

3) Motion (to clarify the semantics of friend declarations of a local class):

Move we:

-- add to 7.3.1.2, paragraph 3, sentence 2, the words "in a non-local class", so that it reads:

If a friend declaration in a non-local class first declares a class or function, the friend class or function is a member of the

innermost
enclosing namespace.

-- add to 11.4, paragraph 2:

If a friend declaration appears in a local class [class.local] and the
name specified is an unqualified name, a prior declaration is looked
up without considering scopes outside the innermost enclosing
non-class scope. For a friend function declaration, if there is no
prior declaration, the program is ill-formed. For a friend class
declaration, if there is no prior declaration, the class that is
specified belongs to innermost enclosing non-class scope, but if
subsequently referenced, its name is not found by name lookup in
the innermost enclosing non-class scope until a matching declaration
is provided in that scope. [Example:

```
class X;
void a();
void f() {
  class Y;
  extern void b();
  class A {
    friend class X;    // okay, but X is a local class, not ::X
    friend class Y;    // okay
    friend class Z;    // okay, introduces local class Z
    friend void a();   // error, ::a is not considered
    friend void b();   // okay
    friend void c();   // error
  };
  X *px;               // okay, but ::X is found
  Z *pz;               // error, no Z is found
}
```

--end example]

4) Motion (to resolve issue 674 and clarify class name lookup and ambiguity in
the presence of using-declarations):

Move we:

-- insert the text shown below into 10.2, paragraph 2
(insertions are indicated between arrows):

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 (1)<==. 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.

         ==> (1) Note that using-declarations cannot be used to resolve
             inherited member ambiguities; see 7.3.3.  <==

      -- add an example following the text above to show how a using-
declaration
         may find a static member:

           [Example:
        struct A { static int i; };
        struct B: A {};
        struct C: A { using A::i; };
        struct D: B, C { void foo(); };
        void D::foo() {
             i;    // finds A::i two ways: as C::i and A::i in B
                   // no ambiguity because A::i is static
        }
           --end example]


5) Motion (to resolve issue 675 and clarify the definition of final
overrider
   in the presence of using-declarations);

Move we:

      -- insert the text shown below into 10.3, paragraph 2
         (insertions are indicated between arrows):

         If a virtual member function vf is declared in a class Base and in
a
         class Derived, derived directly or indirectly from Base, a member
         function vf with the same name and same parameter list as Base::vf
is
         declared, then Derived::vf is also virtual (whether or not it is
so
         declared) and it overrides 88) Base::vf. For convenience we say
that
         any virtual function overrides itself. Then in any well-formed
class,
         for each virtual function declared in that class or any of its
direct
         or indirect base classes there is a unique final overrider that
         overrides that function and every other overrider of that
function.
         The rules for member lookup (10.2) are used to determine the final
         overrider for a virtual function in the scope of a derived
class==>,
         but ignoring names introduced by using-declarations
         [Example:

        struct A {
             virtual void f();
        };
        struct B: virtual A {
             virtual void f();
        };
        struct C: B, virtual A {
```

```
        using A::f;
    };
    void foo() {
        C c;
        c.f();      // calls B::f, the final overrider
        c.C::f();   // calls A::f because of the using-declaration
    }
      --end example]<==.
```

6) Motion (to resolve issue 700 and indicate that a diagnostic is not required
   if a function or object is used and not defined):

Move we:

-- replace the first sentence of 3.2 paragraph 3 with:

   Every program shall contain exactly one definition of every non-inline function that is used in that program; no diagnostic required. Every program shall contain at least one definition of every inline function that is used in that program; no diagnostic required.

-- replace the last sentence of 3.2 paragraph 3 with:

   An object that is used in a program shall be defined and only one definition shall be provided.

7) Motion (to only require that a static data member be defined if it is used
   in a program):

Move we:

-- replace the first three sentences of 9.4.2 paragraph 2 with:

   The declaration of a static data member in its class definition is not a definition and may be of an incomplete type other than cv-qualified void. A definition shall be provided for the static data member if it is used (_basic.def.odr_) in the program. The definition shall appear in a namespace scope enclosing the member's class definition.

-- replace 9.4.2 paragraph 4 with:

   If a static data member is of const integral or const enumeration type, its declaration in the class definition can specify a constant-initializer which shall be an integral constant expression (_expr.const_). In that case, the member can appear in integral constant expressions within its scope. The member shall still be defined in a namespace scope if it is used in the program and the namespace scope definition shall not contain an initializer.

-- replace the first sentence of 9.4.2 paragraph 5 with:

   There shall be exactly one definition of a static data member that is

used in a program; no diagnostic is required, see _basic.def.odr_.

8) Motion (to resolve issue 728 and to clarify how linkage-specifications affect object declarations and definitions):

Move we:

-- insert the text shown below into 3.1, paragraph 2 (insertions are indicated between arrows):

A declaration is a definition unless it declares a function without specifying the function's body (8.4), it contains the extern specifier (7.1.1) or ==> a linkage-specification (footnote 23) (7.5) <== and neither an initializer nor a function-body,

-- replace footnote 23 with:

(footnote 23): Appearing inside the brace-enclosed declaration-seq in a linkage-specification does not affect whether a declaration is a definition.

-- replace the following text in 7.5 paragraph 7:

An object defined within an

```
extern "C" { /* ... */ }
```

linkage-specification is still defined (and not just declared).

with:

The form of linkage-specification that contains a brace-enclosed declaration-seq does not affect whether the contained declarations are definitions or not (3.1); the form of linkage-specification directly containing a single declaration is treated as an extern specifier (7.1.1) for the purpose of determining whether the contained declaration is a definition.

9) Motion (to resolve issues 635 and 725; to render recursive local static initialization undefined and allow early initialization);

Move we:

-- insert the text shown below into 6.7, paragraph 4 (insertions are indicated between arrows):

The zero-initialization (8.5) of all local objects with static storage duration (3.7.1) is performed before any other initialization takes place. ==> A local object of POD type (3.9) with static storage duration (3.7.1) initialized with constant-expressions is initialized before its block is first entered. An implementation is permitted to perform early initialization of other local objects with static storage duration under the same conditions that an implementation is permitted to statically initialize an object with static storage duration in namespace scope (3.6.2) Otherwise such an object <==

**is
initialized the first time control passes through its declaration;
such object is considered initialized upon the completion of its
initialization. If the initialization exits by throwing an
exception,
the initialization is not complete, so it will be tried again the
next
time ==> control enters the declaration. If control re-enters the
declaration (recursively) while the object is being initialized,
the
behavior is undefined.
[Example:**

```
int foo(int i)
{
    static int s = foo(2*i);  // recursive call - undefined
    return i+1;
}
```

-end example] <==

**10) Motion (to clarify temporary creation and clarify lifetime for
temporaries
bound in ctor-initializers and return statements):**

Move we:

-- change the beginning of 12.2 paragraph 1 as follows

**==> OLD TEXT:
While evaluating an expression, it might be necessary or
convenient
for an implementation to generate temporary objects to hold values
resulting from the evaluation of the expression's subexpressions.
During this evaluation, precisely when such temporaries are
introduced
is unspecified.<==**

**==> NEW TEXT:
Temporaries of class type are created in various contexts: binding
an
rvalue to a reference (8.5.3), returning an rvalue (6.6.3), a
conversion that creates an rvalue (4.1), a throw-expression
(15.1),
in a try-block (15.3), and in some initializations (8.5). <==**

**-- change 12.2 paragraph 5 as follows (insertions are indicated
between
arrows)**

**The second context is when a reference is bound to a temporary.
The
temporary to which the reference is bound or the temporary that is
the
complete object to a subobject of which the temporary is bound
persists for the lifetime of the reference or until the end of the
scope in which the temporary is created, whichever comes first. A
temporary holding the result of an initializer expression for a
declarator that declares a reference persists until the end of the
scope in which the reference declaration occurs. A temporary bound
to
a reference ==> member <== in a constructor's ctor-initializer
(12.6.2)**

persists until the constructor exits. A temporary bound to a reference
parameter in a function call (5.2.2) persists until the completion of
the full expression containing the call. A temporary bound ==> to the
returned value <== in a function return statement (6.6.3) persists
until the function exits. In all these cases, the temporaries created
during the evaluation of the expression initializing the reference,

11) Motion (to resolve issue 723 and allow pointer to member casts in pointer
to member constant expressions):

Move we:

-- replace 5.19 paragraph 6 with:

A pointer to member constant expression shall be created using the
unary & operator applied to a qualified-id operand (_expr.unary.op_),
optionally preceded by a pointer to member cast (_expr.static.cast_).

12) Motion (to specify when to check access and ambiguity for deallocation
functions):

Move we:

-- replace the entire paragraph 11 of 12.4 [class.dtor] (which merely
restates rules already in 5.3.5 [expr.delete]) with:

Within the definition of each virtual destructor, the
implementation shall look up and find an unambiguous and
accessible non-placement deallocation function. The
implementation shall perform this check even for implicitly-
defined virtual destructors.

-- replace the first sentence of 12.5 [class.free] paragraph 8 with:

When a delete-expression deallocates an object whose static
type has a virtual destructor, the delete-expression calls the
deallocation function that was found (12.4) by looking up a
non-placement deallocation function in the destructor of the
dynamic type of the object.