Name Lookup Issues and Proposed Resolutions

Doc No: **X3J16/94-0014**
WG21/N0401
Date: **January 21, 1994**
Project: Programming Language C++
Reply-To: Neal M Gafter
gafter@mri.com

1 Introduction

(1) This document proposes clarification or resolutions for all of the open core issues about name lookup.

(2) Issues

Chapter 3:	332	333	365	366			
Chapter 9:	124	403	268	269			
Chapter 10:	271	71	46	354	103	343	382
Chapter 11:	288	418					
Chapter 12:	14	17	79	117	355		
Chapter 13:	152	87	43	140			

2 Clarification of x->A::b and x.A::b (332, 333)

(1) The current Working Paper is not clear, nor is it consistent with the Portland decision. The text "These searches must yield a single type which may be found in either or both contexts." (5.2.4p1) should be "The name must refer to a single declaration of a class type which may be found in either or both contexts."

(2) In fact, the entire paragraph needs to be reorganized. By the time we get to the sentence, above, the referent of "these searches" is not clear. I suggest the following minor reorganization of 5.2.4p1 into multiple paragraphs:

> A postfix expression followed by a dot (.) or an arrow (->) followed by an *id-expression* is a postfix expression. For the first option (dot) the type of the first expression (the object expression) must be class object (of a complete type). For the second option (arrow) the type of the first expression (the pointer expression) must be pointer to class object (of a complete type). The *id-expression* must name a member of that class, except that an imputed destructor may be explicitly invoked for a built-in type, see 12.4.
>
> If the *nested-class-specifier* contains a *template-class-id* (14.2), its *template-arguments* are evaluated in the context in which the entire postfix-expression occurs.
>
> If the *id-expression* is a *qualified-id*, the *nested-class-specifier* of the *qualified-id* is looked up as a type both in the class of the object expression (or the class pointed to by the pointer expression) and the context in which the entire *postfix-expression* occurs. The name, if any, of each class is also considered a nested class member of that class. The name must refer to a single declaration of a class type which may be found in either or both contexts.
>
> If the *id-expression* names a nonstatic data member, the result is the named member of the object designated by the value of the first expression, and it is an lvalue if the class object and the member are lvalues. If the *id-expression* names a static data member, the result is the named member of the class. If the *id-expression* names a (possibly overloaded) non-static function member, the expression can only be used as part of a member function call

> (5.2.2). If the *id-expression* names a (possibly overloaded) static function member, the result is the function.

(3) In addition, the following text from 9.4 should be repeated in (or moved to) 5.2.4:

> When a static member is accessed through a member access operator, the expression on the left side of the . or -> is not evaluated.

3 Scope, Name-lookup, and Uninitialized Bases in a *ctor-initialiser* (71, 79, 117, 271, 355)

3.1 Initializing inherited virtual base classes.

(1) Consider this example:

```
struct A { A(); };
struct B : public virtual A { B(); };
struct C : public B { B(); };
```

(2) now, can the constructor for class C use a *ctor-initializer* to construct its virtual A?

```
C::C : A(), B() {} // Allowed?
```

(3) According the the current draft of the working paper, the answer is no (12.6.2 Initializing bases and members, first sentence):

> Initializers for immediate base classes and for members not inherited from a base class may be specified in the definition of a constructor.

(4) But A is not an immediate base class of C. Therefore, C's constructor may not explicitly initialize the virtual A. However, the compiler is obliged to provide this initialization in the most derived class's constructor, so the virtual base must have a default constructor (12.6.2p2-3):

> The class of a complete object (1.3) is said to be the most derived class for the sub-objects representing base classes of the object. All sub-objects for virtual base classes are initialized by the constructor of the most derived class. If a constructor of the most derived class does not specify a *mem-initializer* for a virtual base class then that virtual base class must have a default constructor or no constructors...

(5) Some implementations appear to grant C the ability to initialize the virtual A explicitly. This is not correct because A is not an immediate base class of C. It would be a simple local change to the WP to allow explicit initialization of inherited virtual base classes. Change the first sentence of 12.6.2 to the following

> Initializers for immediate and virtual base classes and for members not inherited from a base class may be specified in the definition of a constructor.

(6) I do not have a strong feeling one way or the other on this issue, because the workaround without this change is easy (add a virtual A to C's list of base classes). I propose to leave the current wording.

3.2 The name of base classes in a *ctor-initializer*

(1) Consider this example

```
typedef struct A { } B;
struct C : B {
      int B;
      C() : A() {} // allowed?
};
```

(2) That is, is the *ctor-initializer* allowed to name the base class by a name different from that used in the class definition? In what scope is the *identifier* (or *qualified-class-specifier*) of a *ctor-initializer*'s *mem-initializer* evaluated? These questions are not answered in the current WP.

(3) I propose the following resolution, and suggested text to be added to 12.6.2:

> The *identifier* (or *qualified-class-specifier*) of a *ctor-initializer*'s *mem-initializer* in a class's constructor is evaluated in the scope of the class. It must evaluate to a nonstatic data member or the type of a direct base class.

(4) If the committee decides to allow explicit initialization of inherited virtual base classes, this will become

> The *identifier* (or *qualified-class-specifier*) of a *ctor-initializer*'s *mem-initializer* in a class's constructor is evaluated in the scope of the class. It must evaluate to a nonstatic data member or the type of a direct or virtual base class.

(5) And the following should be added for clarification:

> A constructor's *mem-initializer-list* can initialize a base class using any name that designates that base class type; the name used may differ from the class definition.

3.3 Uninitialized classes in a *ctor-initializer*

(1) The expressions in a *ctor-initializer* are evaluated in the scope of the body of the constructor (10.4p9):

> A *ctor-initializer* (12.6.2) is evaluated in the scope of the outermost block of the constructor it is specified for. In particular, it can refer to the constructor's parameter names.

(2) This should be reworded to clarify that it is the *expression* (not the *identifier* or *qualified-class-specifier*) in each *mem-initializer* of the *ctor-initializer* that is evaluated in that context.

(3) Because all of the base classes and members may not have been initialized, these expressions can yield unpredictable results. I suggest making such references explicitly undefined. Add the following text to 12.6.2:

> In a *ctor-initializer*, the effect of calling a non-static member function of a class object that has not been fully initialized is undefined. A class object is fully initialized after its last base class has been constructed. In addition, the value of each nonstatic data member is indeterminate until its construction is complete.

(4) With the following example

```
class A {
      virtual int f();
}

class B : A {
      int x;
      int y;
      int f();
      B() : y(f())        // undefined: B::f called
                          // but B not fully initialized.
            x(y)          // undefined; uses indeterminate
                          // value of y
}

class C
```

4 Derived Classes and Ambiguity (46/275, 87, 152, 333, 354)

(1) The ambiguity in multiple inheritance needs to be clarified. The current text states (10.1.1)

> Access to a base class member is ambiguous if the *id-expression* or *qualified-id* used does not refer to a unique function, object, type, or enumerator.

(2) Unfortunately, this disallows overloading

```
struct A {
      int f(float);
      int f(int);
};
struct B : public A {}
B x;

x.f(int); // error - ambiguous?
```

(3) Certainly this is not what was intended. I do not think there is a corresponding problem with the C compatibility hack.

(4) A second problem is that the rules to help determine if the name “refers to a unique” thing are not clear, and appear elsewhere (10p4):

> Name lookup proceeds from the original class (the named class in the case of a *qualified-id*) along the edges of the lattice until the name is found. If a name is found in more than one class in the lattice, the access is ambiguous (see 10.1.1) unless one occurrence of the name hides all the others. A name B::f hides a name A::f if its class B has A as a base and the instance of B containing B::f has the instance of A containing A::f as a sub-object. The second part of this definition is trivially satisfied when multiple inheritance is not used.

(5) This paragraph doesn’t allow the following:

```
struct A { static int x; }
struct B : public A { ... };
struct C : public A { ... };
struct D : public B, public C {
      f() { x = 2; } // ambiguous
}
```

(6) because the name x is found in the lattice twice. But this is allowed by the definition of ambiguity in 10.1.1 becasue x refers to a unique object.

(7) The check for ambiguity should check for the same declaration. See issue 333. This is clear from (10p4) but not from (10.1.1).

(8) This latter quoted paragraph (10p4) should be moved to section 10.1.1 and reconciled with the first quoted paragraph. We need one (rather than two) definition of ambiguity in access to inherited members. Unfortunately, I did not have time to prepare a proposed replacement text when this paper was written.

5 Individial Core Issues in Numeric Order

5.1 Is the name of a constructor a type name or a function name? (14 / 290)

Section: 12.1 Constructors
Status: active
Requestor: Tom Pennello
Emails: core-1461 core-1463

(1) If it is a function name, does it hide the type name?

(2) Also from Tom Plum and Dan Saks: define "constructor" and its "name" 12.1p11 says "A member function with the same name as its class is called a constructor...." But it's been mentioned that this is incorrect. Is a constructor a member function? What is its name?

Proposed resolution

(3) A constructor's name is not considered a function name; for purposes of name lookup, the constructor "function" is never found. This needs to be clarified in the WP.

(4) Section 2.1 notes that it is possible to call a constructor directly by using the function-style cast syntax.

(5) Although it cannot be named directly, it is possible to take the address of a class's constructor. For a class C, this can be done either by taking the address of a name of the class type (with the usual caveats about taking the address of an overloaded function), or it can be done by taking the address of "C::C" which as a result of injection is a synonym for the class name. The latter syntax is preferred for its readability.

(6) This needs to be added to the WP. This is a substantive change.

5.2 What is the scope where the identifier for a conversion function name is looked up? (17)

Section: 12.3.2 Conversion Functions
Status: active
Requestor: Scott Turner
Emails: core-1479

(1) Part of the name look up issue.

```
obj.operator T() // T is not required to be a member of obj's class
typedef int *T;
struct B {
      typedef char *T;
      operator char* ();
      operator short* ();
      operator int* ();
      operator long* ();
};
struct D : public B {
      typedef short *T;
};
void *foo (D &d) {
      typedef long *T;
      return d.B::operator T(); // which T? which conversion func-
tion?
}
```

Proposed resolution:

(2) Identifiers used in naming a type as part of a conversion function name in an expression are evaluated in the context of the full expression. As a result, it may be necessary to use a name different from that appearing at the point of definition of the conversion operator.

(3) This needs to be clarified in the WP. This is an editorial change.

5.3 Argument matching between static and non-static member function (43)

Section: 13.2 Argument Matching
Status: active
Requestor: Scott Turner
Emails: core-1555, core-1556, core-1558, core-1562

```
struct S {
      static void callee();
      void callee() const;
};
void caller (S &s, const S &cs) {
      s.callee();
      cs.callee();
}
```

(1) Is one preferred over another?

Proposed resolution

(2) Static and non-static member functions are not distinguished for the purposes of overload resolution.

(3) Chapter 13 paragraph 3 notes that a class that has a static and a non-static member function, both with the same signature, is ill-formed. The "const" member function qualifier is irrelevant for this test. The example, therefore, is ill-formed.

(4) This needs to be clarified in the WP. This may be a substantive change.

5.4 Ambiguities for static members in multiple inheritance (46/275)

Section: 10.1.1 Ambiguities
Status: active
Requestor: Ron Guilmette
Emails: core-1566, core-1567

```
struct base {
      static void callee(); // static member
};
struct left : public base { }; // No virtual inheritance
struct right : public base { }; // No virtual inheritance
struct derived : public left, public right { };
derived object;
void caller() {
      object.callee(); // MI ambiguity?
}
```

(1) Or, since there is only one static member, are the declarations above well formed?

(2) From Tom Plum / Dan Saks: clarify "reaching the same object"

(3) 10.1.1p?: Page 203 of the ARM contains a comment about "reaching the same object". There is no corresponding statement in the Working Paper. This comment should be added to the Working Paper.

Proposed resolution

(4) Reaching the same object or enumerator doesn't imply an ambiguity. This issue is explicitly resolved in section 10.1.1 of the existing Working Paper. No change to the WP is required, however, see the section "Derived Classes and Ambiguities" for related recommended changes.

5.5 Should a hidden base class and its members be accessible? (71)

Section: 10.1 Multiple Base Classes
Status: active
Requestor: Martin O'Riordan
Emails: core-1636, core-1638, core-1658

(1) Description: From Mike Miller's list of issues:
WMM7. What is the interpretation when a class is used as both a direct and an indirect base class?

```
class A { ... };
class B : public A { ... };
class C : public A, public B { ... };
```

(2) 1. Should it be possible to access 'C's direct base class 'A' or the members of 'C's direct base class 'A'? There is currently no mechanism to access these members of 'C'.

(3) 2. In an explicit base/member initializer list, the name 'A' refers to which occurence of 'A'?

```
C::C(int i) : B ( i ), A ( i ) { ... }
```

(4) 3. Does 'C' have a default assignment operator? If so, how can it be described?

(5) 4. What if B is defined as followed:

```
class B : public virtual A { ... };
```

(6) Are the answers to questions 1., 2., 3. the same?

Proposed resolution

(7) 1. The question was rhetorical, and answers itself. There is nothing to make this example illegal. There is currently no mechanism to name C's direct base class A. No change to the WP is necessary.

(8) 2. In the *mem-initializer* list of a *ctor-initializer*, names are either type names naming direct base classes, or naming direct members (12.6.2). In the example 'A' names a direct base class. There is no ambiguity. No change to the WP is necessary.

(9) 3. Every class has a default assignment operator if no assignment operator is provided by the programmer (12.8). In this case, it is not possible for the user to write an assignment operator equivalent to the one provided by the implementation. No change to the WP is necessary.

(10) 4. The answers to the above questions do not change if A is made a virtual base class of B, because the inheritance DAG does not change. In this case, the virtual 'A' base class of C (which is distinct from the non-virtual base class 'A' of C) is initialized using a default (or no) constructor (12.6.2). No change to the WP is necessary. However, see also the section "Scope, Name-lookup, and Uninitialized Bases in *ctor-initialisers*" of this paper.

5.6 Scope of uninitialized items and constructor *mem-initializers* (79)

Section: 12.6.2 Initializing Bases and Members
Status: active
Requestor: Ron Guilmette
Emails: core-1588, core-1593, core-1596, core-1601, core-1603, core-1605 core-1606, core-1613

See 12.6.2p4:

```
struct S1 {
      S1 ();
      S1 (const S1&);
      ~S1 ();
};

struct S2 {
      S1 a;
      S1 b;
      int i;
      int func ();

      S2 (int*) : a(b) { } /* OK */
      S2 (float*) : i(func()) { } /* OK */

};
```

(1) In the *ctor-initializer* list for the S2::S2(int*) constructor, the value of the member ‘b’ is used before it has been properly initialized. In the *ctor-initializer* list for the S2::S2(float*) constructor, the member function ‘func’ is invoked for an object which has not yet been fully constructed.

(2) Should the scope rules which apply to *mem-initializers* be modified to mandate that compilers detect cases where a *mem-initializer* expression for a given base or member involves some base or member which (according to the ordering rules given in 12.6.2) will not be fully initialized by the time the *mem-initializer* is actually evaluated?

Proposed resolutions

(3) No, the scope rules should not be modified. Diagnostics of the type suggested are not required by the standard. This is a quality-of-implementation issue. If the standard ever recommends “Warning” diagnostics, this should be high on the list.

(4) No changes to the WP are necessary.

(5) See the section “Scope, Name-lookup, and Uninitialized Bases in *ctor-initialisers*” for related recommendations.

5.7 Hiding of an overloaded function name in a containing scope (87)

Section: 13.1 Declaration Matching
Status: active
Requestor: Ron Guilmette

(1) Section 13.1/1 of the 9/92 working paper notes that: “A locally declared function is not in the same scope as a function in file scope.”

(2) 1. The statement talks about “file scope” when it really ought to talk about “containing scope”. For example:

```
void caller ()
{
      void callee (int, int);
      {
```

```
                void callee (int);
                callee (88, 99); // should be an error
          }
    }
```

(3) should result in an error on the indicated line.

(4) 2. The “commentary” text which followed this statement (in the ARM) was dropped from the working paper. This ARM commentary indicates that base classes act as if they were “containing scopes” (relative to derived classes). In other words, the working paper must clarify that for:

```
struct B { void callee (int, int); };
struct D : public B { void callee (int); };
D d;

void caller ()
{
      d.callee (88, 99); // error
}
```

(5) should result in an error on the indicated line.

(6) 3. How does this rule apply in cases where a base class member function is visible via two (or more) inheritance paths, and where some of the paths cause the base class member function to be hidden and others do not? In other words, the working paper must clarify that for:

```
struct B { void callee (int, int); };
struct L : virtual public B { void callee (int); };
struct R : virtual public B { };
struct D : public L, public R { };
D d;
void caller ()
{
      d.callee (88, 99); // error?
}
```

(7) It is unclear what effects (if any) the dominance rules given in section 10.1.1 would have on this example.

Resolution

(8) 1. This is an editorial issue
2. This issue is closed, the WP has already been updated as requested
3. This is a name look up question

Proposed resolution

(9) (3): Yes, this is an error. This issue is handled explitly, if unclearly, in 10p4:

> Name lookup proceeds from the original class (the named class in the case of a *qualified-id*) along the edges of the lattice until the name is found. If a name is found in more than one class in the lattice, the access is ambiguous (see 10.1.1) unless one occurrence of the name hides all the others. A name B::f hides a name A::f if its class B has A as a base and the instance of B containing B::f has the instance of A containing A::f as a sub-object. The second part of this definition is trivially satisfied when multiple inheritance is not used.

(10) In this case, the name “callee” in L hides the name in B. More clarification is provided in 10.1.1, beginning with the following text and an example:

> When virtual base classes are used, a hidden function, object, or enumerator may be reached along a path through the inheritance DAG that does not pass through the hiding function, object, or enumerator. This is not an ambiguity. The identical use with nonvirtual

base classes is an ambiguity; in that case there is no unique instance of the name that hides all the others.

(11) No change to the WP is necessary. However, clarification would be extremely helpful.

(12) See the section "Derived Classes and Ambiguity" for recommendations.

5.8 Can a hidden base class virtual member be overridden? (103)

Section: 10.2 Virtual Functions
Status: active
Requestor: Mike Miller

(1) From Mike Miller's list of issues:
WMM.8. Is it necessary for a base class virtual member to be visible for a derived class to override it?

```
class A { virtual int f(); };
class B : A { int f(int); };
class C : B { virtual int f(); }; // does f() override A::f?

class A { virtual int f(); };
class B : A { int f; };
class C : B { virtual int f(); }; // does f() override A::f?
```

Proposed Resolution

(2) Visibility is not a factor in overriding virtual members (10.2).

(3) No change to the WP is necessary.

5.9 What is the scope of *mem-initializers*? (117)

Section: 12.6.2 Initializing Bases and Members
Status: active
Requestor: Mike Miller

(1) This is from Mike Miller's list of issues:
10.4p9 currently says:

> A *ctor-initializer* is evaluated in the scope of the outermost block of the constructor it is specified for. In particular, it can refer to the constructor's argument names.

(2) The annotation on p.217 of E&S is not obvious from the Working Paper proper.

> Note however that the base class and the member names are looked up in the scope of the class itself.

```
class X {
      int a;
public:
      X(int);
};
X::X(int a) :a(a) { } // perverse, but legal
```

(3) This should probably be included in the WP as well.

Proposed resolution

(4) In a *mem-initializer* list, the names being initialized are either type names naming direct base classes, or naming direct members (12.6.2). The *mem-initializer* expression is evaluated in the scope of the outermost block of the constructor (10.4). These different rules for looking up the name being initialized and names appearing in the expressions in the initializer cause the subtle distinction clarified in E&S. The current WP is

clear even if the information is not in one place.

(5) No change to the WP is necessary, but a clarification would be helpful. See the section "Scope, Name-lookup, and Uninitialized Bases in *ctor-initialisers*" in this paper for related recommendations.

5.10 Are static member operators allowed? (124)

Section: 9.4 Static Members
Status: active
Requestor: Mike Miller / John Armstrong
Emails: core-493

(1) Should it be possible to overload operators with static member functions?

(2) In particular, should it possible to overload unary operators and binary operators with static member functions?

(3) For example:

```
class complex {
      double re;
      double im;
public:
      complex(real,imag) : re(real), im(imag) {}
      static complex operator -(const complex &cplx) // unary -
            {return complex(-cplx.re,-cplx.im);}
      static complex operator +(const complex &cplx1,
                                const complex &cplx2) // binary +
            {return complex(cplx1.re+cplx2.re,cplx1.im+cplx2.im);}
};
```

Proposed resolution:

(4) No, static member operators are not allowed (13.4).

(5) No change to the WP is necessary. Note that the same effect can be had with either normal member functions, or with global friend functions.

5.11 Do unary versions of overloaded operators hide binary versions? (140)

Section: 13.4.1 Unary Operators
13.4.2 Binary Operators
Status: active
Requestor: Mike Miller / Martin O'Riordan
Emails: core-680

(1) WMM.91. Do unary versions of overloaded operators hide binary versions of those operators and vice versa (similarly for prefix/postfix ++ and --)? (This has implications for both derived/base scopes and member/nonmember operators.) For example:

```
class Base
{
public:
      operator & (); // Unary 'address-of'
};
class Derived : public Base
{
public:
      operator & ( int ); // Binary 'bitwise AND'
```

```
};
Derived dd;
&dd; // Is this valid or not ?
```

(2) The problem is, that the names of both the unary and binary operators '&' are the same, and thus the unary form provided by the base class is hidden by the binary form declared in the derived class. Thus the expression above fails due to not finding a suitable match. Should the binary and unary forms of these operators be considered to have to same name, or some different name?

Proposed resolution

(3) Yes, they have the same "name" so the unary version of an operator can hide the binary version. In this respect, overloaded operator names behave the same as other function names. This should be made explicit in the WP. This is probably an editorial change.

5.12 Overloading across namespaces (152)

Section: 13. Overloading
Status: active
Requestor: Bjarne Stroustrup
Emails: core-1693, core-1696, core-1700

```
// example #1
struct X {
      struct S { S(); };
      static void S();
      void f() {
            struct S a;
            S();
      }
};
```

(1) Does the C compatibility hack apply to class members? Should 9.1 restrict the kind of scope the compatibility hack applies to?

```
 // example #2
struct A { struct X { X(); }; };
struct B { static void X(); };

struct C: A, B {

      void f() {
            struct X a;
            X();
      }
};

// example #3
struct A { static void f(int); };
struct B { static void f(double); };

struct C: A, B {
      void h() {
            f(1); // error: ambiguous ?
            f(1.0); // error: ambiguous ?
      }
};
```

(2) How does this relate to example #2?

Proposed resolution

(3) Example 1: Does the C compatibility hack apply to class members?

(4) Yes; there are not restrictions to the scopes in which it applies.

(5) Should 9.1 restrict the kind of scope the compatibility hack applies to?

(6) No.

(7) No change to the WP is necessary for this example.

(8) Example 2: During name lookup, if a name is found to have different definitions from different base classes, there is an ambiguity and the program is ill-formed (10.1.1). The idiom of using a *class-key* or :: to disambiguate (3.2p2) is only described to apply to a name hidden by an inner scope, and so doesn't apply to this example. I would recommend generalizing the idiom by defining it to apply the usual lookup rules, but ignoring any non-type names. This would make example 2 legal. In any case, a precise formulation of the idiom needs to be added to section 10.4. I suggest the following additional text:

> When evaluating the *identifier* of an *elaborated-type-specifier*, non-type definitions are ignored. Similarly, when evaluating the identifier immediately preceding the :: operator only type and namespace names are considered. For example, a *class-key* may be used to refer to a type from an enclosing scopes that is hidden by a local declaration of the same name, or to disambiguate between an inherited type name in one base class and an inherited data member in another.

(9) Example 3: During name lookup, if a name is found to have different definitions from different base classes, there is an ambiguity and the program is ill-formed (10.1.1).

(10) No change to the WP is necessary. However, see the section "Derived Classes and Ambiguity" of this paper for related recommendations.

5.13 How can enclosing class members be used in nested classes? (268)

Section: 9.7 Nested Class Declarations
Status: active
Requestor: Tom Plum / Dan Saks

(1) clarify restrictions on "uses" in nested class 9.7p1.5: "Except by using explicit pointers, references, and object names, ..." Does this mean "Except for member functions with parameters that are pointers or references to an enclosing class, ..." What does "and object names" mean? What other accesses are possible?

Proposed Resolution

(2) The intent of the text in the WP is to say that, although the names from the enclosing class are visible for the purposes of name lookup, the "dynamic" members of the enclosing class cannot be used except by the usual mechanism of supplying an object (with the "." or "->" operator) of the enclosing class type. I suggest the following words to replace the relevant text in the WP:

> Except by use of the '.', '->', or '&' operators, declarations in a nested class can use only those members from the enclosing class that do not designate nonstatic members.

(3) This is an editorial change.

5.14 Which names from the enclosing local scope can a local class use? (269)

Section: 9.8 Local class declarations
Status: active
Requestor: Tom Plum / Dan Saks

(1) clarify restrictions on "uses" in local class 9.8p1.4: The words, as written, are overly restrictive. Declara-

tions "can use only type names, static variables, extern variables and functions, and enumerators from the enclosing scope". What about integral constants? What about string literals? Should the words say something like "An identifier can be used in declarations ... only if the the identifier is a type name, a static var ... etc".

(2) 92/12/17: This issue also applies to 97p15.

Proposed Resolution

(3) The intent of the text in the WP is to say that, although the names from the enclosing function are visible for the purposes of name lookup, the "dynamic" data cannot be used. I think the quoted text should be rewritten: "can use only names that do not designate automatic objects".

(4) This is an editorial change.

5.15 How are inherited members treated (class layout, access)? (271)

Sections: 10 Derived Class
Status: active
Requestor: Tom Plum / Dan Saks

(1) clarify base's non-inheritable members

(2) 10p1.15 should say "Unless redefined in the derived class, [or unless the members are non-inheritable,] members of a base class can be referred to as if they were members of the derived class." Surely this does not apply to ctors, dtors, or assignment. (?)

(3) If this "clarification" reflects the committee's intent, then it follows from this that access-decls can't apply to the non-inherited members.

(4) 92/12/17: There is a serious portability problem lurking in this rather pedantic-sounding issue. Consider a base B with public constructor(s) which is a virtual private base for C, which is in turn a base for D. The constructors of D are obliged to explicitly invoke B's constructors. How does D obtain access? Some implementations appear to tacitly grant the access according to (unpublished?) special rules. Other implementations require the use of access-decls that mention B::B .

(5) The question of "how does D obtain access" is a second issue.

Proposed Resolution

(6) 1. Every class whose base class has a constructor, destructor, or assignment operator has its own; one is supplied by the implementation if not provided by the programmer. The relevant sections of chapter 12 make this clear. This serves to hide the base class's constructors, destructors, and assignment operator. Section 10 should add text to clarify that constructors, destructors, and assignment operators are not inherited. This is an editorial change.

(7) [John Max Skaller pointed out that classes with reference member should never have compiler-generated constructors, destructors, or assignments, even though this rule is not in the WP. As a result, this change might be considered substantive.]

(8) 2. The inherited virtual base can not be initialized directly because it is not a direct base class (12.6.2). No change to the WP is necessary.

(9) See the section "Scope, Name-lookup, and Uninitialized Bases in ctor-initialisers" in this paper.

5.16 friend functions defined in class, and name lookup (288)

Section: 11.4 Friends
Status: active
Requestor: Tom Plum / Dan Saks

(1) 11.4p52 - friend def'd in class is in lexical scope of class. The "name lookup" question has been clarified somewhat with the new section 9.2.1 in Jun 92 draft, but it's not yet clear whether this clarifies the status of friend functions defined inside a class.

(2) [Note JL:] I don't understand this issue. What is "status" suppose to refer to here?

Proposed Resolution

(3) The current WP seems to be unclear on name lookup in the body of friend definitions.

(4) On the one hand, it says "The function is then inline and the rewriting rule specified for member functions is applied" (11.4p5). This rewriting is "considered to be done after preprocessing but before syntax analysis and type checking of the function definition" (9.3.2p1). In addition, "global friend definitions" were banned from the language (Portland motion 6), so the function being defined could only be a member of some class.

(5) On the other hand, "A friend function defined in a class is in the (lexical) scope of the class in which it is defined. A friend function defined outside the class is not" (11.4p5).

(6) I suggest we drop the sentences "A friend function defined in a class is in the (lexical) scope of the class in which it is defined. A friend function defined outside the class is not." (11.4p5). This substantive change removes a self-contradiction in the WP.

(7) In addition, the WP is unclear on name lookup for the return type and types in the argument list for inline functions. This is a general problem with the "rewriting" applied to inline functions defined in a class, both members and friends. For example:

```
typedef char T;
struct C {
      typedef int T;
      T f() { return 3 };
}
```

(8) if this is rewritten naively, this becomes

```
typedef char T;
struct C {
      typedef int T;
      T f();
}
T f() { return 3; };     // error: wrong return type
```

(9) In what scope are the types named in the return type and parameter list evaluated? Does C::f() return an int or a char? Or is it an error? I suggest that the "rewriting" be clarified to apply only to the body of the function; return and parameter types are bound in the context of the original definition.

5.17 Impact of declarations and definitions on the name look up rules (332)

Section: 3.1 Declarations and Definitions
Status: active
Requestor: Sam Kendall
Emails: core-2002, core-2003, core-2004, core-2010, core-2012, core-2013

(1) This relates to the new name look up rules adopted in Portland.

(2) Example 1:

```
struct A { static int x; }; // 1
struct A; // 2
void f(A a) {
      a.A::x; // well formed?
}
```

(3) The search in A's context finds declaration #1. The search in f's context finds declaration #2. According to the rule, refering to 2 different declarations renders the program ill-formed.

(4) Should name look up look for a definition first before looking for a declaration?

(5) Are line //1 and line //2 refering to the same name?

(6) Example 2:

```
struct S {
      injected S; // magic declaration
      int x;
};
... p->S::x ...
```

(7) S refers to ::S and to S::S, two different declarations: ill-formed.

(8) Should the term "declaration" be replaced by the terms "same meaning"?

Proposed Resolution

(9) Example 1 is ill-formed: "An elaborated-type-specifier with a class-key used without declaring an object or function introduces a class name exactly like a class definition but without defining a class" (9p2). The name "A" is therefore multiply defined. Had line //2 been written "struct A t;", then it would have been clear that this does not introduce the type name "A" but rather refers to the existing type name "A". No change to the WP is necessary.

(10) Example 2 imagines an injected declaration for S distinct from the struct's own declaration. It is in fact the same declaration, so the program is well-formed.

(11) The term "same declaration" should not be replaced by "same meaning".

(12) See the section "Clarification of x->A::b and x.A::b" of this paper for related recommendations.

(13) Editorial WP changes are necessary.

5.18 Should the new name look up rules go through typedefs? (333)

Section: 3.1 Declarations and Definitions
Status: active
Requestor: Tom Wilcox
Emails: core-2005, core-2006, core-2007, core-2008

(1) This relates to the new name look up rules adopted in Portland. Search after '.' and '->' operators must be done in 2 contexts:

(2) o in the class' context

(3) o in the lexical context of the expression

(4) According to the rule, refering to 2 different declarations render to program ill-formed. Should name look up look "through" typedefs to find out if 2 types are the same?

```
typedef struct A { static int x; } A;
void F(A a) { a.A::x; } // ill-formed?
```

(5) Should the searches look for the types that the typedefs represent?

(6) The same question applies to name look up in the following contexts:

(7) o In the reconsideration rule

(8) o In multiple inheritance

```
struct B1 {
      typedef int T;
};
struct B2 {
      typedef int T;
};
struct D: B1, B2 {
      T x;  // error ambiguous or looking through typedefs to find
the
            // the same type?
};
```

(9) Should the searches look “through” typedefs in these situations as well?

Proposed Resolution

(10) The answer is “No”, it should not look “through” typedefs. This example is ill-formed. Similarly, in the reconsideration rule and multiple inheritance, the lookup should not look “through” typedefs.

(11) See also the sections “Clarification of x->A::b and x.A::b” and “Derived Classes and Ambiguity” in this paper.

5.19 Virtual member functions and static member functions (343)

Section: 10.2 Virtual function
Status: active
Requestor: Ron Guilmette

(1) Problem 1:

(2) May a member function which is declared as virtual in a base class be declared as static in a derived class?

```
struct B { virtual int f (); };
struct D : public B { static int f (); }; /* error? */
```

(3) Problem 2:

(4) Conversely, may a member function which is declared as static in a base class be declared as virtual in a derived class?

```
struct B { static int f (); };
struct D : public B { virtual int f (); }; /* error? */
```

(5) (Note that for each of these cases, I am really asking two questions, i.e. (1) “Must a standard conforming processor issue diagnostics for such cases?” and (2) “May a standard conforming program contain such cases?”)

Proposed Resolution

(6) Problem 1:

(7) From WP (10.2p2):

> 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)...

(8) From WP (10.2p5):

> The virtual specifier implies membership, so a virtual function cannot be a global (non-member) (7.1.2) function. Nor can a virtual function be a static member, since a virtual function call relies on a specific object for determining which function to invoke.

(9) These two excerpts render Problem 1 ill-formed. The error is one that must be diagnosed by a conforming processor. No change to the WP is necessary, However, when the WP is made more clear on what errors must be diagnosed, this should be included in the list.

(10) Problem 2: There is no problem with this example. The virtual function hides the static one in the derived class. No change to the WP is necessary.

5.20 Inheriting a virtual function "ambiguously" (354)

Section: 10.1.1 Ambiguities
Status: active
Requestor: Ron Guilmette

(1) Is it permitted for a derived class to inherit a given virtual member function (with a given parameter type profile) either directly or indirectly from two or more base classes? (Some existing implementations disallow this.)

```
struct L { virtual void member (); };
struct R { virtual void member (); };
struct D : public L, public R { }; /* ok? */
```

(2) Assuming that the code shown above is in fact valid in the language, is it permissible to call such an "ambiguously inherited" virtual member function for an object whose static type is one of the base class types, but whose dynamic type is the derived class type? If so, what will be the effects of such a call?

```
void test () {
      L *lp = new D;
      R *rp = new D;
      D *dp = new D;

      lp->member (); /* ok? what effect? */
      rp->member (); /* ok? what effect? */
      dp->member (); /* error - ambiguous call */
}
```

Proposed Resolution

(3) There are no problems with the declarations of L, R, and D. There is nothing in the WP to disallow this, neither should there be. Because the virtual functions are not overridden, the call through lp calls L::member(), and the call through rp calls R::member(). No changes to the WP are required.

(4) The call through dp it is not allowed by the current text of 10.1.1, although it is not very clear. See the section "Derived Classes and Ambiguity" in this paper.

5.21 The dynamic type of an object during construction (355)

Section: 12.6.2 Initializing Bases and Members
Status: active
Requestor: Ron Guilmette

(1) Section 12.6.2/4 of the 9/92 working paper says:

> A *mem-initializer* is evaluated in the scope of the constructor in which it appears.

(2) Also, in 10.4/9 we have:

A *ctor-initializer* (12.6.2) is evaluated in the scope of the outermost block of the constructor it is specified for.

(3) These rules define (in part) how the (static) process of name lookup should be carried out during the compile-time semantic analysis of the initializer expressions given in *mem-initializers*. They do not however shed any light on the question of the "dynamic type identity" of an object during the early part of its construction, i.e. during the dynamic run-time evaluation of the expressions given in *mem-initializer*s.

(4) This separate issue is illustrated by the following example:

```
struct B
{
      int i;
      B (int arg) : i (arg) { }
      virtual int member () { return 0; }
};

struct D : public B
{
      virtual int member () { return 1; }
      D () : B (member ()) { }
};

int main () { D d; return d.i; }
```

(5) For this example, the current working paper fails to specify the dynamic type which the object being constructed should be considered to have at the time when the 'B (member ())' *mem-initializer* is executed. Thus, it is unclear what the behavior of the above code should be. In particular, the invocation of 'member' in the *mem-initializer* for B may result in the B::member function being called or in the D::member being called, or neither of those two. (In fact, for a number of existing implementations, the compiled version of this example merely segfaults and then core dumps... an outcome which seems entirely less than satisfactory.)

Ron Guilmette's proposed resolution

(6) If asked, I would recommend to the committee the following (two-part) resolution of this issue.

(7) First, modify the current verbage in 12.6.2/4 and 10.4/9 to say that name lookup for an expression given in a mem-initializer for a given base class is done from the point of view of the scope of that given base class. (Note that name lookup for expressions given in *mem-initializers* for members of the current class would not change. Only the *mem-initializers* for base classes would be affected.)

(8) This simple rule would insure that members of the current derived class and/or members of other base classes (including virtual members) which have not yet been initialized could not be improperly used during the execution of any particular *mem-initializer* for any given base class (since they would simply not be "in scope" when those *mem-initializers* were compiled).

(9) Second, add verbage to 12.6.2 which would clearly specify that the dynamic type identity of the object being constructed (as regards to virtual functions) during the execution of a given *mem-initializer* is dependent upon which base or member is being initialized by the given *mem-initializer.* For a *mem-initializer* for a given base class, the dynamic type identity of the object during the evaluation of that specific *mem-initializer* is that of the given base class. For a *mem-initializer* for a member of the current class, the dynamic type identity of the object being constructed is always the current class type.

(10) This second set of (dynamic) rules would be fully consistant with the (static) name lookup rules suggested above. They would also serve to fully define the dynamic type identity of the object being constructed during the evaluation of any given *mem-initializer.* (Note that this specific bit of semantics is, at the moment, totally undefined). Finally, this second set of rules would define the dynamic type of objects under construction in a way which is consistant with the effect and intent of other existing language rules. In particular, these rules would insure that during any period of time when a derived type object has not yet had all of its

bases fully initialized, that derived class object will continue to act as if it were only an object of some base class type (at least with respect to virtual functions).

(11) (Footnote: Note that the combined effects of this possible resolution of the issue(s) would be to render the behavior of the program given above "well defined", and it would necessarily exit with a zero exit status under the rules suggested herein.)

Proposed resolution

(12) See the section "Scope, Name-lookup, and Uninitialized Bases in ctor-initialisers" of this paper. This proposal is much less of a change to the defacto language than what Ron recommends.

(13) The *ctor-initializer* for class B in D::D() includes a call to the nonstatic member function "member" of the class D, but D has not been fully initialized. With my recommended changes, the example is therefore undefined.

5.22 Scope of enumerators declared within exception specifications (365)

Section: 3.2 Scopes
Status: active
Requestor: Ron Guilmette

(1) What is the scope of an enumerator whose declaration appears within an enumeration type definition which itself appears within the exception specification part of a function or member function declaration or definition?

```
void foo () throw (enum E1 { red, green, blue });
void bar () throw (enum E2 { cyan, magenta, yellow }) { }
```

Proposed Resolution

(2) Types may not be defined (even as incomplete types) in exception specifications. This is consistent with function parameter and return types (8.2.5p5). This needs to be spelled out in the WP in section 15.5.

(3) This is an editorial change.

(4) See also issue 366.

5.23 Scope of types defined in exception specifications (366)

Section: 3.2 Scopes
Status: active
Requestor: Ron Guilmette

(1) Description: What is the scope of a type whose first declaration appears within the exception specification part of a function or member function declaration or definition?

```
void foo () throw (enum E);
void bar () throw (struct S)
{
      struct S { int mbr; }; // completes previous?
}
```

Proposed Resolution

(2) Types may not be defined (even as incomplete types) in exception specifications. This is consistent with function parameter and return types (8.2.5p5). This needs to be spelled out in the WP in section 15.5.

(3) This is an editorial change.

(4) See also issue 365.

5.24 Are destructors in derived classes "overriding functions"? (382)

Section: 10.2 Virtual Functions
Status: active
Requestor: Ron Guilmette

(1) Section 10.2/1 of the 1/93 working paper says: "If a class BASE contains a virtual function vf, and a class DERIVED derived from it also contains a function with the same name and the same argument types, then a call of vf for an object of class DERIVED invokes DERIVED::vf (even if the access is through a pointer or reference to BASE). The derived class function is said to 'override' the base class function."

(2) Section 10.2/2 says: "An overriding function is itself considered virtual. The 'virtual' specifier may be used for an overriding function in the derived class, but such use is redundant."

(3) Note that the quotation (above) from 10.2/1 says that an "overriding" function (in a derived class) must have the same name as the function which it overrides (in the base class). Thus, a destructor for a derived class never overrides any destructors in any of its base classes (according to the current wording). This in turn also implies that a destructor for a derived class is NOT implicitly virtual, even if the given derived class has one or more (direct or indirect) base classes which have virtual destructors.

Ron Guilmette's proposed resolution

(4) I believe that 10.2/1 needs to be revised so that it explicitly defines the term "overriding function" (used in 10.2/2) and so that this definition explicitly notes that a destructor for a given derived class is an "overriding function" which overrides any and all virtual destructors declared in any of the direct or indirect base classes of the given derived class.

(5) Such semantics seem to be consistant with existing practice.

(6) (It would probably also be useful if 12.4/4 and 12.4/8 contained cross-references to 10.2.)

Proposed Resolution

(7) I fully agree with Ron's proposed resolution. This is an editorial change to the WP.

5.25 Scope injection for type members of anonymous unions (403)

Section: 9.5 Unions
Status: active
Requestor: Ron Guilmette

(1) Are declarations and/or definitions of types which appear within anonymous unions effectively treated as if they had actually been written in the containing scope? (At least one existing compiler provides such treatment.)

```
static union {
      struct S *p1;
};
static union {
      struct S { int member; } *p2;
};
void example ()
{
      p1 = p2; /* ok? types compatible? */
}

static union { typedef int T; };
static union { typedef int T; }; /* error? */
```

Proposed Resolution

(2) The Working paper is clear on this point; the answer is yes (9.5p2).

(3) No change to the WP is necessary.

5.26 Forward friend declaration of members (418)

Section: 11.4 Friends
Status: active
Requestor: Jerry Schwarz (jss@lucid.com)

```
class B ; // incomplete type
class A { friend void B::foo() ; }

void g(B* p) {
      p->foo();   // if friend above is allowed, is this
                  // allowed? That is, does the friend declaration
                  // actually declare B::foo or only have an
                  // effect if B::foo is later declared?
}
```

(1) I have a feeling that this issue was resolved at the time that the “nearest enclosing non-class” rule was adopted, but I can’t find a clear statement in the current working paper.

(2) I have a customer who feels strongly that such declarations should be allowed. Here are some of his comments

> If forward friends cannot be declared, then there is a significant restriction on mutually recursive friends. That is, there is no way to code examples like:

```
class A;
class B {
      ...
      friend A::foo();
};

class A {
      ...
      friend B::fum();
};
```

(3) I personally think that the restriction isn’t “signficant” because you can certainly say “friend A”. There are many kinds of restrictions I would like to express that I can’t with the current version of access control. I don’t think the proposed improvement is worth the headaches that will be created by having to define exactly what forward declarations of class members mean.

(4) At any rate the working paper needs clarification about whether the above is allowed. (And if it is allowed about what it means.)

Proposed Resolution

(5) The text of the WP says: “Function members may be mentioned in friend declarations after their class has been defined.” (9.3p4) “An *elaborated-type-specifier* with a *class-key* used without declaring an object or function introduces a class name exactly like a class definition but without defining a class” (9p2). Since the class is not defined, the example is illegal.

(6) To make it more clear, this text should be added to section (11.4) as well.

(7) This is an editorial change.