Document Number: J16/06-0008 = WG21 N1938
Date: 2006-02-27
Reply to: William M. Miller
Edison Design Group, Inc.
wmm@edg.com

Lookup Issues in Destructor and Pseudo-Destructor References

I. Introduction and Background

Name lookup in explicit destructor calls is complicated by the fact that, like constructors, destructors do not have "names" per se. Instead, an explicit reference to a class's destructor is written using a special syntactic form involving a *type-name* that nominates the class. This *type-name* is not required to be the name of the class but can be a typedef or a template type parameter from the expression context, making lookup of the destructor "name" different from that of any other class member – the member name need not be found in the scope of its class. Another complicating factor is that the destructor may be referred to by a *qualified-id*, meaning that the class name appears twice. Historically, at least, C++ has allowed these two *class-name*s to differ, requiring two lookups.

Pseudo-destructors were added to the language to support generic programming; they allow a function template to use the destructor syntax with a dependent type without having to make a special case for scalar types, which do not have destructors. They add a different wrinkle to name lookup when the destructor syntax is seen, because a scalar type does not define a scope in which the pseudo-destructor "name" can be looked up. By analogy with destructors, a pseudo-destructor reference can involve a *qualified-id*-like form with the *type-name* appearing twice, again potentially requiring two lookups.

The definition of how destructor lookup should be done has evolved over time. For instance, ISO/IEC 14882:1998 contains the following example in 12.4¶12:

```
struct B {
   virtual ~B() { }
};
struct D : B {
   ~D() { }
};

D D_object;
typedef B B_alias;
B* B_ptr = &D_object;

void f() {
   D_object.B::~B();              // calls B's destructor
```

```
   B_ptr->~B();                  // calls D's destructor
   B_ptr->~B_alias();            // calls D's destructor
   B_ptr->B_alias::~B();         // calls B's destructor
   B_ptr->B_alias::~B_alias();   // error, no B_alias in class B
}
```

The different treatment accorded `B_ptr->~B_alias()` and `B_ptr->B_alias::~B_alias()` is illuminating. The fact that `~B_alias()` was to be accepted as an *unqualified-id* in a class member access expression indicates that a dual lookup was intended in this context: the name following the `->` must be looked up both in the class of the object expression and lexically in the context in which the expression occurs. The rejection of the *qualified-id* form, on the other hand, shows that the *unqualified-id* `~B_alias` was to be looked up solely in the scope designated by the *nested-name-specifier*, i.e., handled as an ordinary qualified name lookup.

In contrast, this example from 3.4.3¶5 (also in the 1998 Standard) reflects a later stage in the Committee's thinking:

```
struct A {
   ~A();
};
typedef A AB;
int main()
{
   AB *p;
   p->AB::~AB();       // explicitly calls the destructor for A
}
```

Here it is apparent that the *unqualified-id* in a *qualified-id* is also intended to be subject to a dual lookup and not simply a qualified name lookup, in spite of the explicit qualification.

Issue 244 in the Core Language Issues List was originated to reconcile the discrepancy between these two examples. Discussion of the issue revealed that the Standard did not, in fact, normatively specify how the *unqualified-id* in a *qualified-id* naming a destructor is to be looked up. The resolution of issue 244, adopted by the Committee in October, 2002, replaced the wording in 3.4.3¶5

> In a *qualified-id* of the form:
>
> $::_{opt}$ *nested-name-specifier* `~` *class-name*
>
> where the *nested-name-specifier* designates a namespace scope, and in a *qualified-id* of the form:
>
> $::_{opt}$ *nested-name-specifier class-name* `::` `~` *class-name*
>
> the *class-name*s are looked up as types in the scope designated by the *nested-name-specifier*.

with

Similarly, in a *qualified-id* of the form:

$$::_{opt}\ \textit{nested-name-specifier}_{opt}\ \textit{class-name}\ ::\ \sim\ \textit{class-name}$$

the second *class-name* is looked up in the same scope as the first.

This is the status quo as of this writing as reflected in the current Working Draft, J16/05-0165 = WG21 N1905.

II. Problems with the Issue 244 Resolution

As described in core language issue 399, one major problem with the resolution of issue 244 is that it is not clear what "looked up in the same scope as the first" really means. There are at least four possibilities:

1. The second *class-name* must be declared in exactly the same declarative region as the first.

2. The second *class-name* is looked up normally in the scope in which the first *class-name* was declared. This differs from #1 in that names from containing scopes or base classes would be considered.

3. The second *class-name* is looked up in the class of the object expression if the first was found by the class lookup or in the lexical context of the entire postfix expression if the first was found by the lexical lookup. This differs from #2 in that scopes closer to the class of the object expression or the context of the entire postfix expression would be considered.

4. The first *class-name* is looked up in two scopes, so "looked up in the same scope as the first" means that the second is also looked up in both those scopes.

Another point made in issue 399 is that all of these possibilities save the last beg the question of what happens if the first *class-name* is found by both lookups? Should one context be preferred over the other for looking up the second *class-name*? If so, in a call like `p->C1::~C2()` the look-up would fail if `C2` were found only in the "wrong" context, even though both `C1` and `C2` are declared together in one of the contexts.

Another way to handle finding the first *class-name* in both lookups would be to perform a dual lookup for the second *class-name* as well in this case. This approach (or interpretation #4 of the current wording) raises its own question: if the second *class-name* is found in both contexts, must the two names refer to the same class, or (in the spirit of the currently-proposed resolution of core issue 305) would it suffice if only one of the two names refers to the class of the object expression?

Yet another point raised in issue 399 is that the syntactic requirement for naming a destructor is

"*class-name* `::` `~` *class-name*." Looking up the second name in the same scope as the first could result in finding a name that is not a *class-name* in that scope, even though it is a *class-name* in the other. For example:

```
template<typename T> struct S { };
typedef S<int> SI;

void f(SI* p) {
   p->SI::~S();
}
```

Here `SI` is found (only) by the lexical lookup in the context of the expression, so `S` is looked up there, too. However, in this context, `S` is not a *class-name* but a *template-name*, even though it is a *class-name* (the injected-class-name) in the class of the object expression. Should this call be ill-formed under the issue 244 resolution?

III. Suggested Resolution for Issue 399

All of the problems cited in the preceding section stem from the fact that the second *class-name* in this form of destructor reference is permitted to differ from the first, thus requiring that the second name be looked up; the questions all deal with the specification and/or results of this second lookup. As was noted in the original discussion of issue 244, there is no compelling rationale for allowing these names to differ.

[*Digression:* In fact, assuming that the currently-proposed resolution for core issue 305 is adopted, there will be no need at all for this form of destructor reference, so there is even less reason to support the obfuscatory practice of using different names in the *nested-name-specifier* and the destructor "name." The two-*class-name* form of destructor reference is mostly an historical accident: *The Annotated C++ Reference Manual* required that form in the original specification of explicit destructor call ("Destructors are invoked... explicitly using the destructor's fully qualified name," *ARM* 12.4, p. 278). Very early in the standardization process, it was observed that using a *qualified-id* for the destructor name would suppress the virtual call mechanism and that the unqualified forms `p->~C()` and `x.~C()` presented no special problems. As a result, support for the simpler forms was added, but the fully-qualified form was also retained.

As discussed in issue 305, there remain certain obscure cases in Standard C++ where a *qualified-id* is required to avoid ambiguity between the lexical and class lookups for the *class-name* used in the destructor reference:

```
struct A { };
struct C{
   struct A { };
};

void f(C::A* p) {
   p->~A();
}
```

The lookup of `A` in the class of `p` finds the injected-class-name `C::A`, while the lexical lookup in the context of the expression finds `::A`, resulting in an ambiguity. The proposed resolution of issue 305 addresses this problem by stating that this ambiguity is ignored if either of the two lookups finds a name that refers to the class of the object expression.

With this resolution in place, there is never a need to use the fully-qualified form of destructor reference. In fact, the fully-qualified form should be discouraged to the extent possible because of its effect in suppressing the virtual function call mechanism: if the dynamic type of the object differs from the type used to "name" the destructor, using the fully-qualified syntax leads easily to undefined behavior because the lifetime of a base class object will have been ended without ending that of the derived class object.

For compatibility with existing code, the fully-qualified syntax must continue to be supported, even though not needed, but allowing the two *class-name*s to differ runs counter to the legitimate desire to discourage even the simple form of the fully-qualified syntax. —*end digression*]

As revealed by the record of discussion in issue 244, one possible resolution that was considered was simply to require that the second name be identical to the first and to do no lookup at all on the second name. This approach was rejected in favor of the still-problematic resolution that was ultimately adopted, primarily on the basis of its interaction with *template-id*s. Given the remaining difficulties in the current specification, I think that the no-lookup approach is worth reexamining.

First, I do not believe that the *template-id* examples cited as problematic with the name-matching approach actually are problems. There appear to be two categories of issues:

1. Where both *class-name*s are *template-id*s, do the *template-id*s name the same type?

This is the question behind the example cited in the discussion of issue 244:

```
A<int>* aip;

aip->A<int>::~A<int>();         // should work
aip->A<int>::~A<char>();        // should not
```

This does not appear to cause any difficulties with the same-name approach. 5.1¶7 already requires that

> Where *class-name* `::~` *class-name* is used, the two *class-name*s shall refer to the same class

and the existing rules (14.4) are adequate to determine whether two *template-id*s refer to the same class. As long as the *template-name*s in the two *template-id*s are the same (which is true by definition in this approach), the existing rules are sufficient to handle these cases.

2. What if one *class-name* is a *template-id* and the other one is an *identifier*?

Assuming that the *identifier* in the first *class-name* (either the *class-name* itself or the *template-*

name in the *template-id*) is the name of the template, this concern is addressed by the dual nature of a class template's injected-class-name (14.6.1¶1):

> Like normal (non-template) classes, class templates have an injected-class-name (clause 9). The injected-class-name can be used with or without a *template-argument-list*. When it is used without a *template-argument-list*, it is equivalent to the injected-class-name followed by the *template-parameter*s of the class template enclosed in <>. When it is used with a *template-argument-list*, it refers to the specified class template specialization, which could be the current specialization or another specialization.

The template specialization's injected-class-name will be found in the class of the object expression, so the use of that name in the second *class-name* will also refer to the injected-class-name, allowing it to have or not to have a *template-argument-list*. If there is an explicit *template-argument-list* in the second *class-name*, this is equivalent to the preceding issue where 5.1 and 14.4 assure that the second *class-name* matches the first; otherwise, the bare injected-class-name in the second *class-name* is by definition the same type as the *template-id* in the first.

Suggested wording of 3.4.3¶5:

> If a *pseudo-destructor-name* (5.2.4) contains a *nested-name-specifier*, the first (or only) *type-name* is looked up as a type in the scope designated by the *nested-name-specifier*. If the *pseudo-destructor-name* contains a second *type-name*, no lookup is performed on that name; instead, it shall be the same as the first. Similarly, in a *qualified-id* of the form:
>
> `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name* `::` `~` *class-name*
>
> no lookup is performed on the second *class-name*; instead, the *identifier* (or *template-name* if the *class-name* is a *template-id*) shall be the same as the *identifier* or *template-name* in the first *class-name*. [*Note:* It is not necessary that both *class-name*s be *identifier*s or both be *template-id*s. If the first *class-name* is a *template-id* or refers to the injected-class-name of a class template specialization (14.6.1), the *identifier* or *template-name* in the second *class-name* also refers to the injected-class-name of that class template specialization. The two *class-name*s must denote the same class type (5.1); see 14.4 regarding equivalence of *template-id*s. —*end note*] [*Example:*

```
struct C {
   typedef int I;
};
typedef int I1, I2;
extern int* p;
extern int* q;
p->C::I::~I();              // I is looked up in the scope of C
p->C::~I();                 // I is looked up in the scope of C
q->I1::~I2();               // error: I2 is not the same name as I1
```

```
struct A { };
typedef A AB;
AB* pa;
pa->AB::~AB();              // explicitly calls the destructor for A
pa->AB::~A();               // error: A is not the same  name as AB

template<typename T> struct X { };
X<int>* pxi;
pxi->X::~X<int>();          // X is X<int>, class-names match
pxi->X<int>::~X();          // X is X<int>, class-names match
pxi->X<int>::~X<char>();    // error, X<char> is not X<int>
```

—*end example*] [*Note:* 3.4.5 describes how name lookup proceeds after the `.` and `->` operators. —*end note*]

In addition, the example in 12.4¶12 would need to be changed:

```
B_ptr->B_alias::~B();       // error: B is not the same name as B_alias
```

IV. Namespace-Qualified Destructor and Pseudo-Destructor References

The discussion of core issue 399 contains the following note, dated September, 2004:

> The resolution for issue 244 removed the discussion of `p->N::~S`, where `N` is a *namespace-name*. However, the resolution did not make this construct ill-formed; it simply left the semantics undefined. The meaning should either be defined or the construct made ill-formed.

I do not believe this is an accurate description of the current state of the working draft. While it is true that the explicit discussion of this form was removed from 3.4.3¶5, the general treatment of namespace qualification in 3.4.3.2 still gives the same meaning to this form of destructor reference.

To see why this is true, recall from the introduction of this paper that destructors do not have names; instead, 5.1¶1 indicates that ~ *class-name* is an *unqualified-id*, and ¶7 explains

> A *class-name* prefixed by ~ denotes a destructor; see 12.4.

"Name lookup associates the use of a name with a declaration of that name" (3.4¶1), and "A *name* is a use of an identifier" (3¶4). In a *qualified-id* like `N::~S`, the name to be looked up is `S`, not `~S`, because `~S` is not an identifier and thus not a name.

This is consistent with 3.4.3.2¶1, which says:

> If the *nested-name-specifier* of a *qualified-id* nominates a namespace, the name

> specified after the *nested-name-specifier* is looked up in the scope of the name-space...

Thus, in an expression like `p->N::~S`, the name `S` is looked up in the scope of `N`. If `S` is found to be a *class-name* in the scope of `N`, `N::~S` is an *id-expression* denoting the destructor of the class `N::S` in the class member access expression (5.2.5). This processing is exactly the same as described by the wording excised from 3.4.3¶5 by the issue 244 resolution, with the exception that the original wording made explicit that the name was looked up as a type.

I am not sure why namespace-qualification of destructor references was considered to be needed, but it appears that its inclusion in the 1998 Standard was intentional, as there was explicit provision for it in 3.4.3¶5. Perhaps a clue can be obtained from the discussion found in core issue 305, which deals with examples like:

```
struct A { };
struct C {
   struct A { };
   void f();
};

void C::f() {
   ::A* a;
   a->~A();
}
```

In the current Standard, the reference to `A` in `a->~A()` is ambiguous, and the observation is made in the discussion of the issue, "You can't say `a->~::A()`," which would resolve the ambiguity. However, it is possible in the current Standard to say `a->::~A()`, which means the same thing.

As noted above, if the proposed resolution for issue 305 is adopted, this ambiguity will no longer occur, however, removing this situation as a motivation for namespace-qualification of destructor references.

There is a similar provision for *pseudo-destructor-name*s. The grammar in 5.2¶1 contains the following definition:

> *pseudo-destructor-name:*
>
> $::_{opt}$ *nested-name-specifier*$_{opt}$ *type-name* `::` `~` *type-name*
>
> $::_{opt}$ *nested-name-specifier* `template` *template-id* `::` `~` *type-name*
>
> $::_{opt}$ *nested-name-specifier*$_{opt}$ `~` *type-name*

The third production provides support for pseudo-destructor references of the form `p->C::~T()` and `p->N::~T()`, where `C` is a class, `N` is a namespace, and `T` is a *type-name* denoting a scalar type; `p` is a pointer to that scalar type.

The motivation for such forms is clearer for pseudo-destructors than it is for real destructors. In a reference to a class destructor, the object expression determines a class type in which the *class-name* used in the destructor "name" can be looked up. Assuming the proposed resolution for issue 305 is adopted, it will always be possible to write `p->~C()`, where `p` is a pointer to a class

named `C`, to invoke the destructor with no qualification needed for the class type, regardless of the scope in which the class is defined or the scope in which the reference appears.

That is not the case for a pseudo-destructor, however. In a pseudo-destructor reference of the form `p->~T()`, `T` must be visible in the expression context because `p` is a pointer to a scalar type and thus does not does not determine a scope that can be searched for `T`. If `T` is a *type-name* declared in a namespace or class scope such that it cannot be found by unqualified lookup from the context of the *postfix-expression*, `T` must be qualified in the pseudo-destructor reference. For example, one of the major C++ validation suites contains code similar to the following:

```
namespace N {
   enum E { e };
}

void f() {
   N::E* p;
   p->N::~E();
}
```

Use of the `N::` qualifier allows the *type-name* `E` to be found when forming the pseudo-destructor reference.

(By analogy with destructor references, the "historical accident" described in the digression in section III above is also supported for pseudo-destructor references, e.g., `p->N::E::~E()`. The rationale for this "fully-qualified" form, however, is even less compelling for pseudo-destructors than for destructors, because a scalar type does not define a scope that can be opened using a *nested-name-specifier*: "`E::`" is nonsense if `E` is a scalar type.)

However, this motivation is significantly less persuasive when considered in the context of the intended use for pseudo-destructors: there is no reason ever to invoke a pseudo-destructor on a known type, and in the generic context, the unknown type will always be a template type parameter or a dependent type, never namespace-qualified.

My personal opinion is that, while I see no great motivation for supporting namespace qualification of destructor and pseudo-destructor references, I also see no particularly strong reason for removing them. Given that they are adequately handled in the current working draft, on balance I would suggest leaving them as is with no further changes.

V. Specification Problems for Pseudo-Destructor References

There are a number of problems with the way pseudo-destructors are handled in the current working draft. (Some of these are covered in core issue 555, while others are mentioned only here.)

It is necessary to cover some background information before delineating the various problems. The first point to note is that a destructor reference is an *id-expression*, i.e., either an *unqualified-id* or a *qualified-id*. These are defined in the grammar and descriptive material in 5.1, "Primary expressions." In contrast, a pseudo-destructor reference is neither an *unqualified-id* nor a *quali-*

fied-id; instead, the nonterminal *pseudo-destructor-name* enters the grammar as a special kind of *postfix-expression* and is defined in 5.2, "Postfix expressions." As a result of this distinction, a pseudo-destructor reference is not a class member access expression and thus is not described in 5.2.5, "Class member access," rather, it has its own subsection: 5.2.4, "Pseudo destructor call."

The first problem to note is that determination of which case applies to a given expression is formally circular. 5.2.4¶1 begins,

> The use of a *pseudo-destructor-name* after a dot `.` or arrow `->` operator represents the destructor for the non-class type named by *type-name*.

It then goes on in ¶2,

> The left-hand side of the dot operator shall be of scalar type. The left-hand side of the arrow operator shall be of pointer to scalar type.

5.2.5¶1 begins,

> A postfix expression followed by a dot `.` or an arrow `->`... and then followed by an *id-expression*, is a postfix expression.

It then continues in ¶2,

> For the first option (dot) the type of the first expression... shall be "class object"... For the second option (arrow) the type of the first expression... shall be "pointer to class object"...

That is, to know whether a given expression is a class member access or a pseudo-destructor call, you must determine whether the tokens following the `.` or `->` constitute an *id-expression* or a *pseudo-destructor-name*, which then also places requirements on the postfix expression before the dot or arrow operator. That decision depends on knowing whether the *identifier* in `~T` is a *class-name* (which makes `~T` an *unqualified-id* and thus part of an *id-expression*) or just a *type-name* (which makes it part of a *pseudo-destructor-name*). In order to know that, you must look up the *identifier* – and the lookup rules are different, depending on whether the expression is a *pseudo-destructor-name* or an *id-expression*!

In fact, the intent here is different, avoiding the apparent circularity: the determination of whether a given expression is a class member access of a pseudo-destructor call is based not on the syntax of what follows the `.` or `->` but on the type of the postfix expression preceding the operator. If that left-hand expression has a scalar type before `.` or a pointer to scalar before `->`, the expression is treated as a pseudo-destructor call. If it has a class type or pointer to class type, the expression is a class member access. These two sections need to be restructured to reflect this intent.

Another problem with the specification of pseudo-destructor handling is that some of the lookup rules for pseudo-destructor calls appear to be specified in 3.4.5, "Class member access." Not only is this inappropriate in light of the fact that a pseudo-destructor call is, in fact, not a class member access, the way these rules are written makes them impossible to apply. 3.4.5¶2-3 read as fol-

lows:

> If the *id-expression* in a class member access (5.2.5) is an *unqualified-id*, and the type of the object expression is of a class type `C` (or of pointer to a class type `C`), the *unqualified-id* is looked up in the scope of class `C`. If the type of the object expression is of pointer to scalar type, the *unqualified-id* is looked up in the context of the complete *postfix-expression*.
>
> If the *unqualified-id* is `~`*type-name*, and the type of the object expression is of a class type `C` (or of pointer to a class type `C`), the *type-name* is looked up in the context of the entire *postfix-expression* and in the scope of class `C`. The *type-name* shall refer to a *class-name*. If *type-name* is found in both contexts, the name shall refer to the same class type. If the type of the object expression is of scalar type, the *type-name* is looked up in the scope of the complete *postfix-expression*.

Note that these two paragraphs explicitly apply to "a class member access," which, according to 5.2.5¶2, requires that the object expression be a class type or pointer thereto. Consequently, the statements in 3.4.5¶2-3 that purport to handle object expressions of scalar type are vacuous because an object expression of scalar type can never occur in a class member access. (Not to mention the fact that ¶2 deals only with "pointer to scalar type" and ¶3 only with "scalar type"!)

A similar objection might be advanced regarding the treatment of pseudo-destructor calls in 3.4.3¶5: because a *pseudo-destructor-name* is not a *qualified-id*, it could seem inappropriate to deal with it in a section entitled "Qualified name lookup." Here, however, I believe the criticism to be less justified. The specification in 3.4.3¶5 deals only with *pseudo-destructor-name*s that contain *nested-name-specifier*s, and it is precisely the *nested-name-specifier* in a *qualified-id* that is dealt with in this section, anyway.

Even accepting the intent of the specifications in 3.4.5 and 3.4.3 to deal with *pseudo-destructor-name*s, however, there is still a gap in the coverage. 3.4.5 claims to deal with pseudo-destructor calls of the form `p->~T()`, while 3.4.3 describes the handling of *pseudo-destructor-name*s that contain a *nested-name-specifier*. Nowhere in the current working draft is there any description of how to handle a pseudo-destructor call of the form `p->T::~T()`. (The "`T::`" is not a *nested-name-specifier*; even though the grammar of *nested-name-specifier* has recently changed to allow a non-class *type-name* to appear, the semantic description of 5.1¶7-8 makes it clear that only *class-name*s and *namespace-name*s are permitted in *nested-name-specifier*s.)

The most straightforward means of addressing these structural issues in the specification of pseudo-destructor calls would be to create a new subsection in 3.4 for them, just as 5.2 has parallel subsections for pseudo-destructor calls and class member access expressions. As noted above, however, it probably makes most sense to continue to describe the handling of *nested-name-specifier*s in *pseudo-destructor-name*s in 3.4.3, with a pointer to it in the new subsection.