X3J16/95-0183
WG21/N0783

Namespace Issues and Proposed Resolutions

John H. Spicer
Edison Design Group, Inc.
jhs@edg.com

September 26, 1995

Revision History

Version 1 (95-0183/N0783) – September 26, 1995: Distributed in the pre-Tokyo mailing.

Introduction

This document attempts to clarify a number of namespace issues that are currently either undefined or incompletely specified.

This document is not intended as a formal proposal of any specific language changes. Rather, it is intended as to be used as a framework for discussion of these issues. Hopefully this will ultimately result in formal proposals for language changes.

Organization of the Document

The document is organized in sections. Each section consists of a list of questions. Each question has an answer, a status, the version number of the first version of this document that included the question, and the version number of the last change in the question. This allows the reader to skip over questions that have not changed since the last time he or she read the document.

Summary of Issues

Friend Issues

1.1 Under what circumstances does a friend function declaration refer to a previously declared function? Where are friend functions injected?

1.2 Under what circumstances does a friend class declaration refer to a previously declared class?

1.3 Proposal to allow global qualifiers on friend declarations.

Qualified Lookup Issues

2.1 Clarification of qualified lookup rules.

2.2 Clarification of 1.5 namespace rules in qualified namespace lookup

2.3 Do the new operator lookup rules for namespaces apply to the global namespace?

2.4 Do the new qualified lookup rules apply to globally qualified names?

Other Issues

3.1 Clarification of unnamed namespace semantics.

3.2 Can a using-declaration reference a member of the current namespace?

3.3 Can member using-declarations refer to constructors, destructors, and `operator=` functions?

3.4 Rules for looking up names in declarators.

3.5 How are names looked up in contexts that require a namespace name?

3.6 What is the linkage of members of unnamed namespaces?

Friend Issues

1.1 Question: Under what circumstances does a friend declaration refer to a previously declared function?

Status: Open

`example`

The WP currently says that a friend function that has not previously been declared (in any enclosing scope) declares a function in the nearest enclosing namespace. This means that the meaning of friend declarations within a namespace depends on declarations in the scopes that enclose the namespace.

The following example illustrates that it is essential that a set of friend declarations for a given name should all declare entities in the same scope. If they declare entities in different scopes, one entity will end up hiding others during name lookup.

As illustrated below, once `A::f` is injected, `::f` is hidden and can only be called using explicit global qualification.

```
void f(char);

namespace A {
        class B {
                friend void f(char);  // ::f(char) is a friend
                friend void f(int);   // A::f(int) is a friend
```

```
                void bf();
        };
        void B::bf()
        {
                f(1);  // calls A::f(int)
                f('x');  // also calls A::f(int) because ::f is hidden
        }
}
```

Note: When a friend declaration (in which the declarator is not a qualified name) the function is looked up in a set of scopes. If the function is found, the declaration is considered to refer to that previous declaration. If it not found, the declaration is considered to declare a new function and that declaration is "injected" into some enclosing scope. The issue here is which set of scopes is searched to find a previous declaration.

So, from this example we have our first rule:

> All friend declarations for a given name must declare entities in one particular scope.

So the question is, which scope to they declare entities in? There are two possibilities, either

1. When looking for a previous declaration of the function, look until the nearest enclosing namespace is reached, or
2. When looking for a previous declaration, look in all enclosing scopes for the *name* of the function that was declared. If a previous use of the name is found, the declaration is injected into that scope. If no previous use of the name is found the friend is injected into the nearest enclosing namespace scope.

Rule #2 would mean that the presence of any function called `f` in an enclosing scope, whether or not the types match, would be enough to cause a friend declaration to inject into that scope.

I believe that rule #2 is clearly unacceptable. A friend declaration in a namespace would be affected by *any* global declaration of that name. Consider what this would mean for operator functions! The presence of *any* `operator+` function in the global scope would force all friend `operator+` operators to appear in the global scope too! The presence of a template in the global scope would have the same effect.

After all, the whole idea of namespaces is to isolate code so that it is not affected, and does not affect, code in other namespaces.

Answer: When looking for a previous declaration of a friend function the search terminates at the nearest enclosing namespace scope.

Note: This proposal only concerns friend declarations in which the declarator is not a qualified name.

Version added: 1
Version updated: 1

1.2 Question: Under what circumstances does a friend class declaration refer to a previously declared class?

Status: Open

```
namespace N {
        class A { void f(); };
}

using namespace N;

namespace M {
        class B {
                friend class A;  // Without this rule
                                 // makes N::A a friend
                B();
        };
        class A { void f(); };
}

void N::A::f() { M::B  b; }  // A friend under current rules

void M::A::f() { M::B  b; }  // A friend under proposed rules
```

Answer: As with friend functions, when looking for a previous declaration of a friend class the search terminates at the nearest enlosing namespace scope. If no previous declaration is found, the class is injected into the nearest enclosing nonclass scope.

Note: The behavior of elaborated type specifiers in other contexts would not be affected by this change.

```
struct A;  // Always introduces A into the current scope
struct B* b; // Searches all active scopes for a previous
             // declaration.  Injects in the nearest nonclass
             // scope if none found.
```

Version added: 1
Version updated: 1

1.3 Proposal to allow global qualifiers on friend declarations.

Status: Open

The existence of namespaces in general makes it necessary to sometimes use a global qualifier in friend declarations. This is even more necessary if issue 1.1 is adopted as proposed.

```
void f(int);
void g(int);

namespace A {
        void f(int);
        class B {
                friend void ::f(int);  // can't currently be done
                friend void g(int);    // injects A::g as
                                       // proposed in 1.1
        };
}
```

The proposal is that a global qualifier be permitted on the declarator in friend declarations. If the return type is an identifier, the declarator must be enclosed in parentheses.

```
struct X {};
X f(int);

namespace A {
        class B {
                friend X (::f)(int);
        };
}
```

Version added: 1
Version updated: 1

1.4 Question: Can friend functions be defined using qualified names?

Status: Open

```
class X1 {
        void f();
};

class X2 {
        friend void X1::f(){}  // Already prohibited
};

namespace A {
        typedef int J;
        void f(J);
}

namespace B {
        class C {
                friend void A::f(J) { /* allowed? */ }
        };
}
```

Answer: The equivalent class case is already prohibited to avoid the need to specify the name lookup rules for cases like this (or so I'm told). There is no reason to allow the namespace case. The function must already be declared in the namespace so there is no reason why the definition can't be there as well. Definitions in friend functions are only useful in cases where injection is taking place.

Note: Name lookup is affected because when scanning a declarator in which the declarator-id is a qualified name, the scope of the declarator must be reactivated before scanning the rest of the declarator. For more information, see issue 3.4.

Version added: 1
Version updated: 1

Qualified Lookup Issues

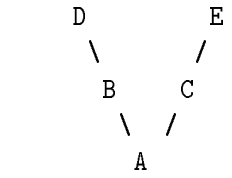
2.1 Clarification of qualified lookup rules.

Status: Open

The rules for performing a qualified lookup in the presence of using-directives need clarification.

The rules make it clear that when you say `A::i` in a program like the following, you stop searching once you've found that there is an `i` in namespace `A`. It is also clear that if a nonfunction of a given name is in both B and C, then the name is ambiguous.

The rules don't make it clear what happens if the names are present at different points in a using directive "hierarchy".




```
namespace E { int l; void f(); }
namespace D { int j, l; void g(double); }
namespace C { int k;    void g(char); using namespace E; }
namespace B { int j;    void g(int);  using namespace D; }
namespace A { int i;    void f();     using namespace B;
                                      using namespace C;}

int main()
{
        int i;
        i = A::i;
        i = A::j;
        i = A::k;
        i = A::l;  // Ambiguous -- D::l or E::l
        A::f();
        A::g(1);   // B::g(int)
        A::g('x'); // C::g
        A::g(2.0); // ambiguous (B::g or C::g.  D::g is hidden)
}
```

The result of a lookup such as `A::j` should be the equivalent of merging the results of a lookup of `B::j` and `C::j`.

The lookup of `B::j` would find the `j` in `B` and would stop there. The lookup of `C::j` would find nothing. So the end result would be `B::j`. `D::j` would not be considered because the lookup would never reach namespace `D`.

Likewise, when constructing an overload set for a lookup of `A::g`, the result would be the merged lookups of `B::g` and `C::g`. The lookup of `B::g` would be the function `B::g(int)`. `D::g(double)` would not be included in the overload set because the lookup of `B::g` would

stop once the `g in \verbB—` is found. The lookup of `C::g` would result in the function `C::g(char). The final result of the \verbA::g—` lookup would be the overload set of `B::g(int)` and `C::g(char)`. Note that `D::g(double)` would not be visible.

Version added: 1
Version updated: 1

2.2 Clarification of 1.5 namespace rules in qualified namespace lookup

Status: Open

In Bjarne Stroustrup's "Relaxation of Qualified Lookup" (N0745R1) the following rules are given to describe the result of the new qualified namespace lookup that follows using directives in the namespace:

> If the set of entities found is empty, the program is ill-formed. If that set has exactly one member, X::m refers to that member. Otherweise, if that set has more than one member, the program is well formed if the use of the name is one that allows a unique member to be chosen, such as overload resolution (13.2 [over.match]) or resolution between class names and non-class names (9.1 [class.name]). Otherwise, the program is ill-formed.

The issue concerns the sentence that says "or resolution between class names and non-class names".

The wording from Bjarne's paper permits a nontype name from one namespace to hide a class name from another namespace. This is in conflict with a clarification that was decided upon by the core-3 group in Monterey for nonqualified lookup.

The rationale for the decision in Monterey was that 3.3.6 [basic.scope.hiding] defines the class/nonclass hiding rule as:

> A class name (9.1) or enumeration name (7.2) can be hidden by the name of an object, function, or enumerator declared in the same scope.

The discussion in core-3 endorsed the position in editorial box 35 which says "The original namespace proposal indicated that the C struct hack does not apply across namespaces. A rule in this subclause should make this more explicit:"

In the following example the two different `i`'s are defined in different scopes so one cannot hide the other, except by the new qualified lookup rules:

```
namespace A {
        int i;
}

namespace B {
        struct i {};
}

namespace C {
        using namespace A;
        using namespace B;
        int j = sizeof(i);  // Ambiguous
        int k = sizeof(C::i); // okay by rules from N0745R1
}
```

The qualified namespace lookup rules should be modified so that the class/nonclass name hiding only applies to names in the same scope, as is the case everywhere else in the language.

When this was posted on the reflector, Bjarne commented in core-6077:

> My intent when I wrote the wording above was for X::m to obey "the usual rules" as defined in 9.1. There was no intent to extend the scope (sic) of the structure tag hack. I think the two sizeof()s in the example should behave identically. Outlawing both is fine with me and - given Motion 15 - editorial (as far as I can see).

Version added: 1
Version updated: 1

2.3 Question: Do the new operator lookup rules for namespaces apply to the global namespace?

Status: Open

13.3.1.2 [over.match.expr] says:

> For a type T whose fully-qualified name is `::N1:...::Nn::C1:...::Cm::T` where each `Ni` is a namespace name and each `Ci` is a class name, the fully-qualified namespace name `::N1:...::Nn` is called the namespace of the type `T`. To look up `X` in the context of the namespace of the type `T` means to perform the qualified name lookup of `::N1:...::Nn::X` (13.3.1.1.1).

I believe this needs to be clarified to correctly handle classes defined in the global scope. Global classes need to have the new namespace operator lookup rules applied too. Otherwise, global types used within other namespaces (that also have that operator defined) will fail to find the correct operators.

```
struct A { };

A operator+(A,A);

namespace B {
        struct B {};
        B operator+(B,B);
        void f()
        {
                A a1,a2;
                a1+a2;  // only works with new operator lookup rules
        }
}
```

We need to add something along the lines of the following:

> For a type `T` whose fully-qualified name is `::C1:...::Cm::T` where each `Ci` is a class name, to look up `X` in the context of the namespace of the type `T` means to perform the qualified name lookup of `::X`.

Note that the new rules, when applied to global scope classes, can result in changes in behavior. But the new behavior is consistent with how the same code would behave if defined in a namespace, so I think this is what we want.

An example where the behavior changes is as follows:

```
struct A {};

A operator+(A, double);

int main()
{
        A a1;
        A operator+(A, int);

        a1 + 1;
        a1 + 1.0; // now calls operator+(A, double), previously called
                  // operator+(A, int);
}
```

Version added: 1
Version updated: 1

2.4 Question: Do the new qualified lookup rules apply to globally qualified names?

Status: Open

The new rules for qualified lookup in a namespace specify that if the name is not found in the namespace, and the namespace contains using directives, that the name should be sought in the namespaces named in the using directives.

Does this rule apply to the global namespace?

```
namespace N {
        int i;
}

using namespace N;

int j = i;   // okay
int k = ::i; // but is this okay?
```

Answer: Yes.

Version added: 1
Version updated: 1

Other Issues

3.1 Clarification of unnamed namespace semantics.

Status: Open

The WP defines the semantics of an unnamed namespace as being equivalent to:

```
namespace UNIQUE {
        // namespace body
}
using namespace UNIQUE;
```

This is incorrect because it makes the code in an unnamed namespace dependent on whether the code is in an original namespace or a namespace extension.

```
namespace {}  // If you remove this line, the
              // use of ::f below is invalid
namespace {
        void f()
        {
                using ::f;
        }
}
```

The WP should be changed to define an unnamed namespace as being equivalent to:

```
namespace UNIQUE {}
using namespace UNIQUE;
namespace UNIQUE {
        // namespace body
}
```

Version added: 1
Version updated: 1

3.2 Question: Can a using-declaration reference a member of the current namespace?

Status: Open

There is no reason to allow this kind of usage. An error would benefit users as the presence of such a using-declaration is most certainly a mistake.

```
namespace A {
        void f();
        using A::f;  // allowed?
}
```

Answer: No.

Version added: 1
Version updated: 1

3.3 Question: Can member using-declarations refer to `operator=` functions, constructors, and destructors?

Status: Open

Answer: Member using-declarations are used to affect the way inherited members may be used in a base class. Constructors and destructors are not inherited it seems unnecessary to permit them in using-declarations. Furthermore, if they were permitted we would have to specify what such using-declarations would mean. When the issue came up on the reflector there were a variety of different semantics that such declarations were assumed to have.

`operator=` can be used in a using-declaration, but the copy assignment operator of the base class will not be included in the set of functions brought in from the base class. In other words, if the base class has only a copy assignment operator, a using-declaration

that refers to the `operator=` for the base class is ill-formed. If the base class has a set of overloaded `operator=` functions, the copy assignment operator will not be included in the set brought into the derived class.

Version added: 1
Version updated: 1

3.4 Rules for looking up names in declarators.

Status: Open

In Monterey, the core-3 group discussed whether or not an entity declared in a namespace could be defined using a name that did not fully specify the qualifier, but rather relied on a using directive to implicitly provide part of the qualified name. For example:

```
namespace A {
        namespace B {
                void f1(int);
        }
        namespace C {
                void f1(int);
        }
}

using namespace A;

void B::f1(int){}  // okay
void C::f1(int){}  // okay
```

it was decided by the group that this was permitted and that there was no reason to make a change.

The group did not discuss a somewhat similar case that can now arise as a consequence of the new qualified namespace lookup rules that were adopted in Monterey.

This was discussed on the reflector, and there seems to be agreement that the new qualified namespace lookup rules must not be used when looking up the final component of a declarator that is a qualified-name.

To understand why this is necessary, recall that when scanning a function declarator such as

```
void A::f(X,Y,Z)
```

the scope of `A` is "reactivated" before the rest of the declarator is scanned. In other words, the remaining names in the declarator (`X`, `Y`, and `Z`, in this example) are looked up in the scope of `A`. Clearly, in order to look names up in the scope of `A`, you need to know what scope you are dealing with when you hit the "`::`".

In the case below, the type of `T` in the definition of the function cannot be determined because you don't know which scope to look in.

```
namespace A {
        namespace B {
```

```
                typedef int T;
                void f1(T);
                void f2(T);
        }
        namespace C {
                typedef char T;
                void f1(T);
                void f2(T);
        }
        using namespace B;
        using namespace C;
}

void A::B::f1(T){}  // okay
void A::C::f1(T){}  // okay

// In a declarator, you need to look up all names after the :: in
// the scope of the qualifier that precedes the ::.  If the new
// qualifier lookup were allowed in the declarator, you can't
// know what scope the qualifier refers to until you've determined
// the type of the function being declared!

void A::f2(T){}  // error
void A::f2(T){}  // error
```

Answer: When looking up the final component of a qualified-id in a declarator the final component of the qualified name must name an immediate member of the class or namespace specified by the qualified-name.

Version added: 1
Version updated: 1

3.5 Question: How are names looked up in contexts that require a namespace name?

In other words, in contexts in which a namespace name is required (using-directives and namespace alias definitions) are only namespace names considered in the lookup?

Status: Open

```
namespace N1 { }

namespace M {
        int N1;
}

using namespace M;

using namespace N1;  // okay or ambiguous?

namespace my_N1 = N1;  // okay or ambiguous?
```

Answer: Only namespace names are considered (i.e., the examples above are okay).

Version added: 1
Version updated: 1

3.6 Question: What is the linkage of members of unnamed namespaces?

Status: Open

When namespaces were first introduced, members of unnamed namespace were said to have no linkage. Later, as a "simplification", this was changed so that members of unnamed namespaces have external linkage, you just don't know their name, so you can't really refer to them from anywhere else.

Although this simplified some things, I think it complicates a number of others, and I think we should go back to the original description.

Unnamed namespaces are supposed to be the replacement for static variables and functions (but not static members). One of the important benefits of static entities is that they are local to a translation unit and don't clutter the externally visible namespace. The new definition of the unnamed namespace eliminates this advantage. Worse, it makes it necessary for implementations to come up with names for things that are guaranteed to be unique – something that is very difficult to do.

Many of the problems arise when using templates in unnamed namespaces.

```
template <class T> struct A {
        void f(T);
};

namespace {
        class B {};
        A<B> ab;  // #1
};
```

Under the original rules (where members of unnamed namespaces had no linkage), declaration #1 was illegal, because as specified in 3.5 [basic.link], names that have no linkage cannot be used to declare entities with linkage. Under the new rules, this is allowed, but what is the mangled name of `A<B>::f(B)`?

Assume that some other file defines the member function f(T) as

```
template <class T> void A<T>::f(T) { /* ... */ }
```

Generating an instantiation of this function requires that information about `<unnamed>::B` be passed from the translation unit in which it was defined to the context in which the instantiation is done. So the things in unnamed namespaces have to be in the externally visible namespace and have to be given unique names, thus eliminating one of the important properties of static entities.

Answer: Members of unnamed namespaces have internal linkage.

The changes needed to make this change are:

- Specify that entities that have external linkage when declared in named namespaces have internal linkage when declared in unnamed namespaces. (Entities that have

internal or no linkage in named namespaces retain that linkage in unnamed namespaces)

- modify the rule in 3.5 to say

  A name with no linkage or internal linkage shall not be used to declare an entity with external linkage. A name with no linkage shall not be used to declare an entity with internal linkage.

Version added: 1
Version updated: 1