

Doc No: SC22/WG21/N1893
J16/05-0153
Date: 2005-10-12
Project: JTC1.22.32
Reply to: Herb Sutter
hsutter@microsoft.com

A Modest Proposal: Fixing ADL (revision 1)

ADL is widely known to be an essential feature. On the other hand, it is also unarguably a source of many subtle problems.

This paper first reviews the motivation behind ADL. It then summarizes characteristic examples of the real problems that arise in practice with ADL, shows their common root cause in that ADL is overeager, and proposes a minor refinement to ADL that I believe solves all of the reported problem cases, and does so with little or no change to the standard library (even though it's still all in one namespace).

The proposed change can cause some existing code to fail to compile:

- Usually all that is necessary is to add a qualification (or `using`) at the call site.
- To preserve the ability to use some library code organization styles, there is a complementary proposal by Daveed Vandevoorde for a facility to explicitly name the current associated namespaces of a type or object in special cases (e.g., a closed source base) when lookup using the C++98 ADL rules is desired.

1. Background

1.1. Why ADL

ADL is a result of having a language that supports both namespaces and nonmember functions, especially when some functions that clearly are part of a type's public interface (e.g., `operator<<`) are required to be nonmembers.

Specifically, ADL exists to treat nonmember and member name lookup more uniformly. The principle behind ADL is that if you call member functions they naturally "come along" with an object for name lookup:

```
o.f();           // look into the scope of o's type, no "using" needed
```

and nonmember functions can be equally part of the interface and should be treated the same way:

```
f(o);           // look into the namespace containing o's type, no "using" needed
```

because some functions on a type must be nonmembers (notably operators `<<` and `>>`) and clearly ought to be first-class members of the type's interface. Languages like C# and Java don't have this problem only because there are no nonmember methods. ADL exists because people realized that the nonmember function parts of a type's interface ought to "come along for the ride" and be naturally usable just as the member functions already come along for free (because of the `.mf()` and `->mf()` syntax which implies the scope of the object's type).

So ADL is all about ensuring that, just as members "come along" without special namespace qualification (because you allude to the scope of the `this` argument with the `o.f()` syntax), so nonmembers that are equally part of the delivered interface of a type should come along for free without special namespace qualification (because you allude to the scope of the argument via ADL in the `f(o)` case).

There is a clear need for nonmember function names to be looked up in the scope of an argument's type's declaration. The canonical example is that otherwise the usual "Hello, world" wouldn't compile, because the required `operator<<` is declared in namespace `std`:

```
// Example 1
//
#include <iostream>

int main() {
    std::cout << "Hello, world" << std::endl;    // error, without ADL
}
```

Absent ADL, the choices to fix Example 1 would be: a) to qualify the `<<` call which defeats the purpose of operator overloading to get natural infix notation; b) to write a using-declaration for `std::operator<<` which would pull in all the other `<<` functions as well (e.g., the ones that work on iterators); or c) to write a using-directive for `std` (some people still really detest the latter, and although I don't agree with that I do think it would be embarrassing if that was the only reasonable way to make this code work).

For more discussion on why ADL is desirable, see [Sutter 2000] Items 31-34. In particular, the key guiding principle in those Items is the following:

"The Interface Principle

For a class X, all functions, including free functions, that both

(a) "mention" X, and

(b) are "supplied with" X [i.e., are in the same namespace as X]

are logically part of X, because they form part of the interface of X."

For example, clearly the `std::operator<<` template that takes `basic_string` parameters, which cannot be a member function, is nevertheless just as much a part of the supplied interface of `basic_string` as are `basic_string`'s member functions `operator+`, `find`, etc. They are all part of the algebra of operations that "come in the box" of things that you can do with `basic_string` objects. Equally clearly, not every nonmember function that happens to mention `basic_string` is part of the interface; for example, a random function `void f(std::string)` in a client's namespace is not part of the supplied interface of `basic_string`. (See [Sutter00] for further justification of the IP.)

It is because ADL is valuable, and because of the principle underlying it, that [Sutter/Alexandrescu 2005] includes the Item:

“57. Keep a type and its nonmember function interface in the same namespace.”

1.2. Why Not ADL

But it is because ADL is overeager that [Sutter/Alexandrescu 2005] also includes the Item:

“58. Keep types and functions in separate namespaces unless they’re specifically intended to work together.”

This is one of the things programmers have to learn today to steer clear of ADL problems. This paper will point out that Item 58 above exists to avoid the problem that this paper would resolve.

The error in ADL is that it pulls in all functions in the namespace of the type, rather than only the functions that were part of the interface of the type (i.e., those that specifically mention the type). When ADL looks up too much, unintended and unwanted names are found, causing us to select the wrong function (in the case when the function found via ADL is a better match, which usually happens with unconstrained templates) or to give deeply mysterious error messages.

2. Problem Examples

2.1. Unspecified behavior for simple code: N1691, Example 1

This example is based on [Abrahams 2004]:

```
// Example 2.1
//
#include <vector>

namespace user {
  class Customer { };

  typedef std::vector<Customer*> index;

  // copy an index, selecting deep or shallow copy for Customers
  void copy( index const&, index&, int deep ) { }

  void g( index& x ) {
    copy( x, x, false ); // legal? not specified by the standard
  }
}
```

Because the argument `x` has a type declared in namespace `std`, ADL reaches over and considers `std::copy` — and `std::copy` happens to be a better match. Additionally, whether ADL will find `std::copy` or not depends on whether `<vector>` includes `<algorithm>`, which is left up to the implementation. I view it as embarrassing that Example 2.1 is nonportable code.

Under this paper's proposal, the Example 2.1 code would be well-specified and call `user::copy`.

Note: This example could also be fixed by using `enable_if` to constrain `std::copy` to take only iterators. The following example cannot be fixed this way.

2.2. N1691, Example 2

[Abrahams 2004] also contained this example, which also unintentionally looks up an unconstrained template via ADL, and again finds it to be a better match:

```
// Example 2.2
//
#include <tr1/shared_ptr>
#include <tr1/tuple>

namespace communications {
    class Channel;
}

namespace user {
    typedef std::tr1::shared_ptr<communications::Channel> pipe;

    // Connect p2 to the output of p1
    void tie( pipe const& p1, pipe& p2 );

    void g( pipe p1, pipe p2 ) {
        tie(p1, p2); // oops, calls tr1::tie
    }
}
```

(Note that here the constness of the reference is sufficient to make `user::tie` a worse match.)

Note: We are not aware of a way to fix this example by using `enable_if` to constrain `tr1::tie`, because `tuple` template parameters (which `tie` is meant to match) must be unconstrained.

Under this paper's proposal, the Example 2.2 code would be well-specified and call `user::tie`.

2.3. From the Real World

This example from `comp.lang.c++.moderated` is similar to Example 2.4, but more subtle.

```
// Example 2.3
//
#include <vector>

namespace N {
    struct X { };

    template<typename T>
    int* operator+( T , unsigned )
        { static int i ; return &i ; /* just to stub in the function body */ }
}
```

```
int main() {
    std::vector< N::X > v(5);
    v[ 0 ];
}
```

Question: Will this program compile? Answer: Maybe yes, maybe no, and when it fails it fails for different and unrelated reasons on different combinations of compilers and standard library implementations.

First, just to acknowledge it and get it out of the way: Is this the programmer's fault (pilot error)? After all, it's true that he did write an unconstrained template. But, well, so does namespace `std`... therefore either he's in good company, or it's the pot calling the kettle black. (He did put the operator template into his own namespace, after all, so he's clearly trying to be a good citizen and play nice.) So no, we can't just blame the programmer here.

I'll demonstrate the problems by extracting code and showing analysis for both libstdc++ 3.2 and Dinkumware 3.13, and why they fail (always or sometimes) and why. Let's first look at an edited extract from libstdc++ 3.2:

```
// Example 2.3(b): An extract from libstdc++ 3.2, distilled and edited
//
namespace __gnu_cxx {
    template<typename T> class __normal_iterator {
    public:
        __normal_iterator operator+(const long& __n) const {...}
    };
}

namespace std {
    template <class T> class vector {
    public:
        typedef __gnu_cxx::__normal_iterator<T> iterator;
        iterator operator[](unsigned __n) { return iterator() + __n; }
    };
}
```

In this case, we can lay the blame squarely at the feet of ADL. This is a true case of "ADL Bites," because under the current rules `N` is an associated namespace of `__normal_iterator<N::X>`, so `N::operator+<N::X>(unsigned)` should be considered, and it happens to be a better match.

So is it the library implementer's fault? Yes, the library implementer could have written the code more defensively. But why should he have to? (This goes to the heart of Abrahams' argument in [Abrahams 2004].) I think it would be hard to seriously lay the main blame at the library writer's feet here.

But what about a library that *is* written to avoid ADL? From Dinkumware 3.13:

```
// Example 2.3(c): An extract from Dinkumware 3.13, distilled and edited
//
namespace std {
    template<class T> struct _Ranit { };
}
```

```

template<class T> class vector {
public:
    class iterator : public _Ranit<T> {
    public:
        iterator operator+(int) const { return iterator(); }
    };
    iterator operator[](unsigned _Pos) { return iterator()+_Pos; }
};
}

```

Compiling this code under various compilers produces various results. This is a case of “Compiler Bugs Bite,” because `N` is *not* an associated namespace of `std::vector<N::X>::iterator` (the `_Ranit<T>` base doesn’t matter). But all shipping versions I tried of Gnu g++ (up to 3.4.3) and Microsoft VC++ (up to 7.1) do incorrectly find it. Comeau 4.3.3 and the current beta of Microsoft VC++ 8 are two compilers that correctly do not find it.

An even simpler variant:

```

namespace std {
    template<class T> class vector {
    public:
        class iterator { // removed inheritance from _Ranit<T>
        public:
            iterator operator+(int) const { return iterator(); }
        };
        iterator operator[](unsigned _Pos) { return iterator()+_Pos; }
    };
}

```

This workaround prevents incorrect ADL on some of the incorrect compilers, but not others which even here still consider `N` an associated namespace. So there are two different compiler bugs at work here: Incorrectly looking into the namespace of a base class template argument, and incorrectly looking into the namespace of an enclosing class template argument.

Interestingly, Example 2.3 compiled fine on other compilers and standard library implementations, including overlaps with the above-mentioned compilers and libraries. Why did it work there? Here’s a sampling:

- **STLport, libcomo, Borland’s Rogue Wave version:** In these implementations, when I tried them, `vector<T>::iterator` is just `T*`, so the operation is `T*+val`, not `iter+val`. Why does that make everything okay? Because ADL still looks into namespace `N`, but `N::operator+<T*>(T*,unsigned)` is not found because it doesn’t exist – you can’t replace the builtin `operator+` for pointers, so a general `operator+` template won’t generate it! (Note: Always using raw pointer iterators isn’t a “solution” ... there are important benefits to having iterators of class type, e.g., “safe STL” iterator checks.) But surely “because you don’t replace built-in operators” is an *extremely* subtle reason to rely on for why these implementations didn’t break on this example.

- **Metrowerks:** Metrowerks’s `vector<T>::iterator` can be configured to be many things, including `T*` and `__debug_iterator<vector<T>, T*>`. But `operator[]` always does “`data() + n`” (not `iterator() + n`) – so again that’s always `T*+val`, not `iter+val`.

Clearly, neither the standard nor the implementations have been in great shape when it comes to ADL. Those that fail, fail for various unrelated and inconsistent reasons. Those that work, work for exceedingly subtle reasons and only narrowly escape falling over the brink. This isn’t a great place to be.

The good news is that this is easy to fix in the standard, and that the fix in the standard will make it easier (not harder) for implementations to conform correctly.

Under this paper’s proposal, the Example 2.3 code would be well-specified and would not consider the user-supplied template in namespace `N`.

2.4. Pointers as iterators

Most people agree that ADL looks up too much, as already shown.

Some people, however, feel that ADL doesn’t look up *enough*. Specifically, pointers (which can be used as iterators) have no associated type and therefore can’t trigger ADL like a full iterator type can do. This is an inconsistency between pointers, which are supposed to be iterators over arrays, and other iterators. For example:

```
// Example 2.4 (a): Today’s situation
//
// code in some other namespace:
int a[10], b[1000];
int *astart = &a[0], *aend = &a[6];
std::copy( astart, aend, &b[0] ); // qualification IS required
std::istream_iterator<int> istart(std::cin), iend;
copy( istart, iend, &b[0] ); // qualification IS NOT required
```

With pointers, `std::copy` is not looked up by ADL because a pointer has no associated namespaces. With `std::istream_iterators`, `std::copy` is looked up by ADL.

I agree this is inconsistent behavior between pointers and UDT iterators, but I believe that the problem is not that pointers don’t trigger ADL, but that UDT iterators *do* trigger ADL; that is, ADL lets you get away with not qualifying `std::copy` — it would be consistent to require that a call to `std::copy` be qualified (`copy` is after all a general-purpose function template, and not merely a part of the interface of a specific iterator type).

For one thing, in the same example with *container* iterators, the programmer must qualify the call to have portable code:

```
std::deque<int> d;
std::deque<int>::iterator dstart = d.begin(), dend = d.end();
std::copy( dstart, dend, &b[0] ); // qualification IS required
```

With pointers, `std::copy` is not looked up by ADL because a pointer has no associated namespaces. With `std::deque<T>::iterator`s, `std::copy` is looked up by ADL *only if* `deque<T>::iterator` is a type defined in namespace `std` (it is not always; see for example Example 2.3(b) about libstdc++); otherwise, qualification is still required. This means that, today, portable code must qualify the above example.

Alternatively, if the previous example instead used a `vector`, then the developer doesn't know whether he has to qualify it (because `vector::iterator` might be either a `std::` class type or just a pointer), and this can change just from going from a debug to a release build:

```
std::vector<int> v;
std::vector<int>::iterator vstart = v.begin(), vend = v.end();
???:copy( vstart, vend, &b[0] );           // qualification MIGHT BE required
```

Also, notice that qualification is already required not only for pointers but also for every user-defined iterator type (unless the container is instantiated with a type from `std::`):

```
My::Container<int> m;
My::Container<int>::iterator mstart = m.begin(), mend = m.end();
std::copy( mstart, mend, &b[0] );           // qualification IS required
```

So this is already unpredictably inconsistent to the user, and the problem is that ADL still pulls in too much, giving unwarranted “convenience” to the standard library's own iterator types in namespace `std` (and to any iterators a vendor might also add to `std`), and only sometimes.

This paper's position is that the desirable outcome would be:

```
// Example 2.4 (b): Under this proposal...
//
// code in some other namespace:
int a[10];
int *astart = &a[0], *aend = &a[6];
std::copy( astart, aend, &b[0] );           // qualification required
deque<int> d;
deque<int>::iterator dstart = d.begin(), dend = d.end();
std::copy( dstart, dend, &b[0] );           // qualification required
My::Container<int> m;
My::Container<int>::iterator mstart = m.begin(), mend = m.end();
std::copy( mstart, mend, &b[0] );           // qualification required
```

Under the proposed change, the qualification would be uniformly required in all the above cases.

2.5. A caution about namespaces and helpers

Note that a library implementation (including an implementation of the standard library) might choose to define helper functions, possibly templates, inside its own namespace. Because today these can easily be picked up via ADL whenever users of the library use a library type from the same namespace, today the only way for a library to have “really private” helpers is to put them in their own

nested namespace, and specifically one that contains no types. With the proposed change, most such helpers would automatically be made immune from inadvertent lookup, as the library author intended.

3. Non-Solutions

3.1. “Operator << should be global”

In response to Example 1, one claim I’ve heard is that we should have designed the standard library differently, specifically that:

“If operator<< were defined as global this would cease to be a problem.” —[Anon]

There are several serious problems with that, but pointing out one is sufficient: As illustrated in real-world code in sections 2.1-3, a global `operator<<` will be hidden by any other `operator<<` in a namespace, with the result that code in that namespace (or in further nested scopes such as classes inside that namespace) cannot use the global operator without qualification. That is unacceptable because the whole point of operator overloading is to facilitate the use of the operator using natural operator notation.

For example, this option fails to work in the simplest case:

```
// assume the standard library put its operator<< functions
// here in global scope, instead of in namespace std
...
namespace N1 {
    // provide a class X that happens to be streamable
    class X { ... };
    std::ostream& operator<<( const X&, std::ostream& );
    // now write a function...
    void f() {
        std::cout << "Hello, world" << std::endl;    // error, global operator is hidden
    }
    // the programmer probably gets a weird error message that
    // there's no conversion from const char[13] to N::X... huh?!
}
```

Besides, putting more names in the global namespace is pollution, and if this were to be our recommendation then namespaces really would have failed (which I don’t believe is true at all).

4. Proposal

4.1. Description

In short, the key proposed change is to not look up just any function in an associated namespace, but to look up only those that could exactly match the argument that triggered the ADL in the first place

(i.e., that have a parameter type that is the same as, or an accessible base of, the argument type, in the same parameter position).

There are two things that need to be fixed:

- **(Desirable) Narrow the set of associated namespaces:** The current set is far too broad. Following the Interface Principle, we only need to perform lookup starting in the namespaces of the type itself and the namespaces of its bases. (Why would we look into the namespaces of template arguments? Nothing in there could possibly be part of the interface of the type, unless the template arguments were also base classes or something. For example, when looking for the nonmember functions that are part of the interface of a template instantiation `C<X>`, we expect to find them in the namespace of `C` (or of `C<X>`'s bases), not in the namespace of the random type `X` someone happened to instantiate `C` with.)
- **(Necessary) Narrow the set of functions looked up:** When an argument of type `T` triggers ADL, then instead of looking up any function in the associated namespaces that happens to be lying around having the correct name, look up only the functions that also have a parameter type of (possibly cv-qualified) `T` or accessible base of `T`, of array of `T`, or of reference to `T` or accessible base of `T`, in the parameter position of the `T` object that triggered the ADL. For example, the expression `std::cout << std::endl` would look into namespace `std` for a function named `operator<<` but consider only those free functions that have a first parameter of type `std::ostream` (or bases thereof, and possibly array-, reference-, or cv-qualified).

Frankly, just making the second change would probably fix every known problem (i.e., it's unlikely that some randomly selected namespace would also happen to harbor a callable function with exactly the same name as the function being looked up *and* having exactly the type we're passing in the function call in the correct position in its parameter list), but we should make the first change too while we're at it to correctly implement the Interface Principle (besides, it's simpler to implement correctly).

One reviewer of a draft asked why ADL should give preference to functions in the type's namespace. In principle, I view it as the right thing to say that nonmembers in the same namespace as the type (and therefore should be authored by the same person/team) get preferential treatment, for the same reason we give member functions special treatment and don't offer a language mechanism to add new member functions to a class the way some languages do.

4.2. Compatibility

As shown in Examples 2.1 through 2.3, this change fixes code that people are trying to write today, but that is an error (or unspecified and nonportable) according to the current definition of ADL.

4.2.1. Calling standard algorithms

One example where the proposed change would break code that is currently correct is when a user is relying on ADL to get a function that is actually unrelated to (i.e., not part of the interface of) the types he's passing. To me this seems like something the user is just getting away with anyway, and the simple fix is to qualify the function name to explicitly reach over into the other namespace. For example:

```

#include <vector>

int main() {
    std::vector<int> v1, v2;
    some_algorithm(v1, v2);    // programmer would have to qualify
}                             // with std:: to reach into std

```

We could optionally also preserve the meaning of such code, by supplying an overload (not specialization) of `some_algorithm` for `vectors` in namespace `std`. In most cases, exactly such an overload is already there; for example, there is already such an overload of `swap` for `vectors` which explicitly makes `swap` part of the interface of `vector`.

There is some evidence that requiring qualification would not be surprising to users. From `comp.lang.c++.moderated`, April 8, 2005, come a note from a user who was surprised that the qualification wasn't required:

```

#include <vector>
std::vector<int> v;
transform(v.begin(), v.end(), v.begin(), Delete_ptr);

This code compiles even though I forgot #include <algorithm> and using std::transform;

```

4.3. Overlaps with other proposals

4.3.1. Operator namespace proposal

Consider the following example supplied by Daveed Vandevoorde. One library A organizes its types into several nested namespaces, and places an algorithm that operates on a smart pointer

```

namespace LibA {
    namespace SubLib1 {
        template<class T> struct N1 {};
        template<class T> void algo(std::shared_ptr<N1>);
    }
    namespace SubLib2 {
        template<class T> struct N2 {};
        template<class T> void algo(std::shared_ptr<N2>);
    }
}

```

Then the user of library A passes an `N1` or an `N2` to another library B:

```

namespace LibB {
    template<class T> void gen(T x) {
        std::shared_ptr<T> p = ...;
        algo(p);    // relies on today's ADL to find either SubLib1::algo or SubLib2::algo
    }
}

```

```

...
N1 n1;
N2 n2;

LibB::gen( n1 );    // ok in today's ADL, error under this proposal
LibB::gen( n2 );    // ok in today's ADL, error under this proposal

```

Today's ADL looks up the two `algo` function template specializations.

Vandevoorde's proposal suggests a `namespace` operator facility that directs lookup to be performed in the associated namespaces (by today's definition) of an object or a comma-delimited list of objects and/or types. This would allow LibA to be organized as above without change, and for the same effect of the above code to be restored by writing `LibB::gen` as

```

namespace LibB {
    template<class T> void gen(T x) {
        std::shared_ptr<T> p = ...;
        namespace(T)::algo(p);
    }
}

```

or as

```

namespace LibB {
    template<class T> void gen(T x) {
        using namespace(T);
        std::shared_ptr<T> p = ...;
        algo(p);
    }
}

```

4.3.2. Unified call syntax proposal

There have been proposals to add a unified call syntax, whereby for example `f(a,b)` could potentially call a member `a.f(b)` or nonmember `f(a,b)` (yes, adding such lookup is problematic because it could break code, but it's worth considering and it keeps being suggested for reasonable reasons including to make it easier to write more flexible templates that aren't sensitive to whether a function or operator that could be either a member or a nonmember is one or the other).

If such a proposal is ever accepted, then it will be even more important to have consistent treatment of members and nonmembers. I say that because clearly when you say `a.f(b)` you want lookup in the namespace where `a`'s type is defined, and the meaning should be the same if you use the syntax `f(a,b)`. This is what ADL is all about, except that ADL picks up more than just functions matching the argument types, which is the problem that I'd like to fix.

4.4. Proposed wording

Modify 3.4.2 as shown (~~strikeout~~ means removed text, and underline means inserted text):

- 1 When an unqualified name is used as the *postfix-expression* in a function call (5.2.2), other namespaces not considered during the usual unqualified lookup (3.4.1) may be searched, and in those namespaces, namespace-scope friend function declarations (11.4) not otherwise visible may be found. These modifications to the search depend on the types of the arguments ~~(and for template template arguments, the namespace of the template argument)~~.
- 2 For each argument type T in the function call, there is a set of zero or more associated namespaces and a set of zero or more associated classes to be considered. The sets of namespaces and classes is determined entirely by the types of the function arguments ~~(and the namespace of any template template argument)~~. Typedef names and using-declarations used to specify the types do not contribute to this set. The sets of namespaces and classes are determined in the following way:
 - If T is a fundamental type, its associated sets of namespaces and classes are both empty.
 - If T is a class type (including unions), its associated classes are: the class itself; the class of which it is a member, if any; and its direct and indirect base classes. Its associated namespaces are the namespaces in which its associated classes are defined.
 - If T is an enumeration type, its associated namespace is the namespace in which it is defined. If it is class member, its associated class is the member's class; else it has no associated class.
 - If T is a pointer to U or an array of U, its associated namespaces and classes are those associated with U.
 - If T is a function type, its associated namespaces and classes are those associated with the function parameter types and those associated with the return type.
 - If T is a pointer to a member function of a class X, its associated namespaces and classes are those associated with the function parameter types and return type, together with those associated with X.
 - If T is a pointer to a data member of class X, its associated namespaces and classes are those associated with the member type together with those associated with X.
 - If T is a class template specialization its associated namespaces and classes are the namespace in which the template is defined; and for member templates, the member template's class; ~~the namespaces and classes associated with the types of the template arguments provided for template type parameters (excluding template template parameters); the namespaces in which any template template arguments are defined; and the classes in which any member templates used as template template arguments are defined.~~ [*Note*: non-type template arguments do not contribute to the set of associated namespaces. —*end note*]

In addition, if the argument is the name or address of a set of overloaded functions and/or function templates, its associated classes and namespaces are the union of those associated with each of the members of the set: the namespace in which the function or function template

is defined and the classes and namespaces associated with its (non-dependent) parameter types and return type.

- 3 If the ordinary unqualified lookup of the name finds the declaration of a class member function, or a block-scope function declaration that is not a using-declaration, the associated namespaces are not considered. Otherwise the set of declarations found by the lookup of the function name is the union of the set of declarations found using ordinary unqualified lookup and the set of declarations found using ordinary unqualified lookup starting in the namespaces associated with the argument types. [*Note*: the namespaces and classes associated with the argument types can include namespaces and classes already considered by the ordinary unqualified lookup. —*end note*] [*Example*:

```
namespace NS {
    class T { };
    void f(T);
    void g(T , int );
    void h(T , float );
}
NS::T parm ;
void g(NS ::T , float );
void h(NS ::T , int );
int main () {
    f( parm );                // OK: calls NS::f
    extern void g(NS ::T , float );
    g(parm , 1);              // OK: calls NS::g(NS::T, intfloat)
    g(parm , 1);              // OK: calls g(NS::T, int)
}
```

—*end example*]

- 4 When considering an associated namespace, the lookup is the same as the lookup performed when the associated namespace is used as a qualifier (3.4.3.2) except that:
- Any *using-directive* s in the associated namespace are ignored.
 - Any namespace-scope friend functions declared in associated classes are visible within their respective namespaces even if they are not visible during an ordinary lookup (11.4).
 - Any function that does not have a parameter type of (possibly cv-qualified) T or accessible base of T, array of T, or reference to T or accessible base of T, in the parameter position of T, is ignored.

5. Acknowledgments

Thanks to Dave Abrahams, Howard Hinnant, John Spicer, Thorsten Ottosen, Daveed Vandevoorde, and others for their very helpful comments on drafts of this material.

6. References

[Abrahams 2004] D. Abrahams. "Explicit Namespaces" (ISO/IEC JTC1/SC22/WG21/N1691 = ANSI/INCITS J16/04-131).

[Anon] Private correspondence.

[Sutter 2000] H. Sutter. *Exceptional C++* (Addison-Wesley, 2000).

[Sutter/Alexandrescu 2005] H. Sutter and A. Alexandrescu. *C++ Coding Standards* (Addison-Wesley, 2005).