Doc No: SC22/WG21/N1792
J16/05-0052

Date: May 1, 2005

Project: JTC1.22.32

Reply to: Herb Sutter
hsutter@microsoft.com

A Modest Proposal: Fixing ADL

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 why ADL is a good idea. That's important, because some critics believe ADL is "a bad idea to begin with" [Anon], completely wrongheaded in concept in the first place. They're wrong; ADL is simply a necessary 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.

This paper 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 without any 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, and I will assert that such code is either broken already (i.e., that it compiled was probably a bug, not a feature) or should probably already have been using a qualified name (and the only change required is to add the missing qualification).

1. Background

1.1. Why ADL

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:

```
// Example 2
//
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).

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 (e.g., std::operator<<(std::cout, "Hello, world");) which defeats the purpose of operator overloading; 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 precisely 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 precisely 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 paper will point out that Item 58 above exists precisely and only to avoid the one problem with ADL that this paper would resolve. This paper's proposal would render Item 58 above obsolete, and I consider that a good thing.

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). In Example 2, you don't want to look up all f's, only those that operate specifically on O's type. That is precisely where the approximation diverges from the Interface Principle, and where the problems with ADL come from.

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

[Abrahams 2004] started with the following example.

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

namespace user {
  typedef std::vector<Customer*> index;

  // copy an index, selecting deep or shallow copy for Customers
  void copy( index const&, index&, bool deep );

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

The analysis in [Abrahams 2004] was that, 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.

That can be true, but there's more: The standard doesn't say at all what the effect of this code will be, because 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 deeply 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`.

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 tr1::shared_ptr<communications::Channel> pipe;

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

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

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

2.3. Hiding: Hinnant's example

Howard Hinnant once sent me an example similar to 2.3, showing the problem isn't just better matches:

```
// Example 2.3
//
#include <utility>
#include <cstdio>

typedef std::pair<int, int> Coordinate;

unsigned distance( Coordinate, Coordinate );

namespace N {
  void f() {
    Coordinate c1 = ..., c2 = ...;
    distance( c1, c2 );                   // legal? not specified by the standard
  }
}
```

This causes problems depending on whether `<algorithm>` is also included (separately by the programmer, or indirectly via `<utility>` which is unspecified). If `<algorithm>` is included, ADL finds `std::distance` to be a candidate

Under this paper's proposal, the Example 2.3 code would be well-specified and call ::distance.

2.4. Toward hilarity: [Sutter00] Item 34, Example 1

This is actual user code from the field:

```
// Example 2.4
//
// In some library header:
//
namespace N { class C {}; }

int operator+( int i, N::C ) { return i+1; }

// A mainline to exercise it:
//
#include <numeric>

int main() {
  N::C a[10];
  std::accumulate( a, a+10, 0 );              // legal? not specified by the standard
}
```

Will the above code compile? Again, the standard doesn't say, which is awful.

On one popular compiler, the above code generated the following clear and helpful error:

```
'class std::reverse_iterator<`template-parameter-1',`template-parameter-2',`template-parameter-3',`template-parameter-4',`template-parameter-5'> __cdecl std::operator+(template-parameter-5,const class std::reverse_iterator<`template-parameter-1',`template-parameter-2',`template-parameter-3',`template-parameter-4',`template-parameter-5'>&)' : could not deduce template argument for 'template-parameter-5' from 'int'

binary '+' : no global operator defined which takes type 'class N::C' (or there is no acceptable conversion)
```

The programmer was completely befuddled, because he hadn't mentioned reverse_iterator anywhere. This is the kind of thing that makes people think about switching to Java. Or assembler.

I analyzed this code six years ago, but here in summary again is the reason. Inside std::accumulate we find code like this:

```
namespace std {
  template<class Iter, class T>
  inline T accumulate( Iter first, Iter last, T value ) {
    while( first != last ) {
      value = value + *first;              // ***
      ++first;
    }
    return value;
  }
}
```

The program is actually calling std::accumulate<N::C*,int>. In the marked line, the expression value + *first causes the compiler to look for an operator+ that takes an int and an N::C (or parameters that can be converted to int and N::C). Even though there is exactly such an operator+(int,N::C) at global scope, the problem is that the compiler may or may not be able to see the global one—depending on what other functions have already been seen to be declared in namespace std at the point where std::accumulate<N::C*,int> is instantiated.

In short, whether the user's code will compile depends entirely on whether this implementation's version of the standard header <numeric>: a) declares an operator+ (any operator+, suitable or not); or b) includes any other standard header that does so. The implementation that produced the above error message happened to have a <numeric> that transitively included an operator+ on reverse_iterators. Bang.

Granted, the programmer is partly at fault here: The programmer should have obeyed the advice in [Sutter/Alexandrescu 2005] Item 57 to "keep a type and its nonmember function interface in the same namespace." If he had done that by putting the operator also into namespace N, ADL would also have found his operator and it would have been a better match, no sweat.

But although the programmer gets part of the blame, it's still awful that our standard doesn't even say whether this program compiles or not. Under this paper's proposal, the standard would guarantee that the code will compile and call ::operator+, although the user still ought to put the operator into the namespace of the type anyway instead of littering the global namespace. (Although this paper's change would let Example 2.4 work as written even with the programmer's operator+ not in namespace N, see section 3.1 for potential problems with the strategy of putting these operators into the global namespace; it's still not recommended because it makes them liable to plain old scope hiding.)

Under this paper's proposal, the Example 2.3 code would be well-specified and call ::operator+(int,N::C).

2.5. Toward abject dismay: A final killer example, on the road to perdition

This example comes straight from comp.lang.c++.moderated. It is similar to Example 2.4, but it's a bit more subtle.

```
// Example 2.5
//
#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 it 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:

```
// 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 be 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:

```
// 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 2.5 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 do so for completely unrelated reasons. Those that work escape the brink for exceedingly subtle reasons. 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 2.5 code would be well-specified and would not consider the user-supplied template in namespace N.

2.6. 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 2.6 (a): Today's situation
//

// code in some other namespace:

int a[10];
int *astart = &a[0], *aend = &a[6];
std::copy( astart, aend, ostream_iterator<int>( cout, " " ) );      // qualification required

vector<int> v;
vector<int>::iterator vstart = v.begin(), vend = v.end();
copy( vstart, vend, ostream_iterator<int>( cout, " " ) );     // qualification NOT required
```

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

With std::vector::iterators, std::copy is looked up by ADL because a pointer has no associated namespaces.

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 typ).

In particular, notice that qualification is already required not only for pointers but also for every user-defined iterator type:

```
My::Container<int> m;
My::Container<int>::iterator mstart = m.begin(), mend = m.end();
```

```
std::copy( mstart, mend, ostream_iterator<int>( cout, " " ) );   // qualification required
```

So the inconsistency really 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).

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

```
// Example 2.6 (b): Under this proposal…
//

// code in some other namespace:

int a[10];
int *astart = &a[0], *aend = &a[6];
std::copy( astart, aend, ostream_iterator<int>( cout, " " ) );      // qualification required

vector<int> v;
vector<int>::iterator vstart = v.begin(), vend = v.end();
std::copy( vstart, vend, ostream_iterator<int>( cout, " " ) );      // qualification required

My::Container<int> m;
My::Container<int>::iterator mstart = m.begin(), mend = m.end();
std::copy( mstart, mend, ostream_iterator<int>( cout, " " ) );   // qualification required
```

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

2.7. 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, 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.

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 section 0, 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 the argument type, in the same parameter position).

There are two things that need to be fixed:

- **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 on earth 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.)
- **Narrow the set of functions looked up:** Instead of looking up any function that happens to be lying around having the correct name, look up only the ones that actually have that type (or base thereof, and possibly pointer-, reference-, or cv-qualified) in the parameter position of the argument that triggered the ADL. For example, the expression std::cout << "Hello" 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 pointer-, 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).

4.2. Compatibility

As shown in Examples 2.1 through 2.5, 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.

I believe the only time 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;
  swap( v1, v2 );        // programmer would have to qualify std::swap to reach into std
}
```

We could optionally also preserve the meaning of such code, by supplying an overload (not specialization) of swap for vectors in namespace std. That explicitly makes swap part of the interface of vector.

4.3. 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, int~~float~~)
    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, pointer to T or accessible base 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, Daveed Vandevoorde, and others for their comments on a draft of this paper presented at the Lillehammer meeting.

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).