Simplifying Argument-Dependent Lookup Rules

Document number: N3595
Authors: Peter Gottschling, SimuNova & TU Dresden
Date: 2013-03-15
Project: Programming Language C++, Evolution Working Group
Reply to: Peter.Gottschling@simunova.com

1 Motivation

ADL rules as defined today are not always intuitive to me — and I dare to postulate: for Joe Coder neither. Especially, when unintended overloads for common function names (e.g. "size") are accidentally found, they can hide the intended overload and require the programmer to write much more code to get the right overload called. I speak from experience. David Abrahams is fighting since a long time that programmers have more control over ADL: [Abr04, Abr12] and many other documents. Other modifications were proposed by Herb Sutter [Sut06].

2 Proposals

2.1 Don't search in template arguments' namespaces!

The following simple program in the Matrix Template Library and similar ones caused a lot of trouble to compile:

```
int main()
{
    mtl::vector::dense_vector<boost::rational> v(12);
    std::cout << "size is " << size(v) << '\n'; // should print 12
    return 0;
}
```

The problem with this program is that the function "size" is searched in the namespaces "mtl::vector" and "boost". This already caused many ugly error messages because the name "size" is used in several boost libraries in an entirely different manner than in MTL.

Another example is the conjugate of a matrix conj(A) that causes ambiguities on some compilers since the function "conj" is defined both in the namespace of the matrix and in std.

A function that is applied on a container should be defined in the namespace of this container — even if the container is defined in std and the elements are in user namespaces. It is possible that a function on the container utilizes functions in the namespace of a template argument but this happens inside the function by referring an object whose type is the template argument. For instance in the example above, an element of the vector given by v[3] has type boost::rational and a call to say f(v[3]) would search "f" in namespace boost. Thus, whenever function overloads in namespaces of the template arguments are needed, one only needs to create according objects. More precisely, overloads in template arguments' namespaces are only relevant when dealing with objects of according types and not when dealing with objects of the template class. For short, there is no reason to search functions in the namespaces of the template arguments.

Asking many programmers and the C++ reflector, we have never heard of a convincing use case for it. The only practical use case I was told was by Eric Niebler for establishing the right initialization order in lambda-like EDSLs. However, this seems to be only the workaround for another problem that should be fixed somewhere else. Nonetheless, we will propose nonetheless a feature to deal with such corner cases.

2.1.1 Full ADL

As stated before, we are not aware of many use cases. Therefore, it seems reasonable to adapt those few classes where function calls should search for overloads in namespaces of template arguments (instead of annotating thousands of classes to turn of the undesired exhaustive lookup). This could be achieved with an according attribute:

```
template <typename T, typename U>
class [[full_adl]] foo {}
```

Or more explicitly on single arguments:

```
template <typename [[full_adl]] T, typename U>
class foo {}
```

In the last example, the function overloads are only searched in the namespace of the first argument but not in the second.

2.2 ADL with Explicit Arguments

Performance can be tuned with meta-programming techniques by providing extra arguments how a function is performed. Unfortunately, such tuning parameters turn of ADL. For instance, a dot product shall be unrolled for the sake of performance and the programmer is free to give the block size of unrolling:

```
alpha= dot(v, w); // #1 works nicely
alpha= dot<8>(v, w); // #2 ADL is turned of
```

Intuitively, the both calls are alike, just that the second one is parametrized. In contrast to the previous section where ADL searches too broadly, here the search is turned off entirely without an obvious reason.[1]

For function calls where a part of the template parameters are explicitly nominated and others are deduced from function arguments, the function overload should be searched in the namespaces of the latter — in the example above v and w.

2.3 Don't treat Inline Friends as 3rd-class Citizens

The following two implementations are supposed to be equivalent:

[1]**Todo:** Or is this a fault of the compilers?

```
template <typename Value> // #1
class vector
{
    // ...
    friend inline size_type size(const self& v) { return v.s; }
};

template <typename Value> // #2
class vector
{
    // ...
    template <typename VV>
    friend typename vector<VV>::size_type
    size(const vector<VV>&);
};

template <typename Value>
typename vector<Value>::size_type
inline size(const vector<Value>& v)
{ return v.s; }
```

Obviously, the second implementation is perceivably more verbose and I clearly prefer the first one. However, there are differences in the lookup: inline friends are looked up very late and often overloads with not as good parameter matches are picked if they are implemented as free functions.

Thus the programmer's decision between #1 and #2 has to take into consideration in which environment a function is called and how likely it is that the name is used somewhere else. Searching in namespaces of arguments (cf. §2.1) complicates matters further. Often, an inline friend implementation works well for years until somebody uses a new library or a library that inter-operated for years causes a conflict since a new function was added.

2.4 Explicitly Turning off ADL

Certain C++ experts turn ADL off by putting the function name into parentheses:

```
(f)(x, y, z); // Don't search f in the ns of x, y, z
```

Needless to say that this is not particularly intentional and over 99 % of C++ programmers will not figure out the impact of the parentheses. Searching the standard document for the regular expression "\(.*\)" is not promising to understand the impact of the parentheses either.

Therefore, we suggest an attribute for disabling ADL:

```
[[no_adl]] f(x, y, z);
```

Here, the intention is much clearer. Even if it is not understood, the programmer has something to search for.

Furthermore, one could turn off ADL more selectively — for instance:

```
f([[no_adl]] x, y, [[no_adl]] z);
```

This form of function call would not add the namespace of x and z to the search space. However, the function "f" would be searched in the namespace of y.

3 Backward Compatibility

Without a prototype implementation it is difficult to foresee all ramifications of these ADL rules but we suspect no significant compatibility issues. Merely, some exotic classes would need an additional [[full_adl]] attribute.

4 Summary

We suggest the following modifications to ADL:

- Do not search in namespaces of template arguments.
- Lookup in the arguments' namespaces can/must be requested explicitly with the attribute [[full_adl]].
- Explicit template arguments do not turn off ADL.
- Inline friend functions are treated with the same priority as free functions.
- Provide an attribute for turning off ADL.

We are convinced that these rules simplify the programming efforts of many programmer — especially those of large projects.

References

[Abr04] David Abrahams. Explicit namespaces. Technical Report N1691=04-0131, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, September 2004.

[Abr12] David Abrahams. ADL control for C++. Technical Report N3490=12-0180, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, October 2012.

[Sut06] Herb Sutter. A modest proposal: Fixing ADL (revision 2). Technical Report N2103=06-0173, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, October 2006.