# Solving the Lookup Problem for Non-member Functions in Presence of Namespaces

Rodrigo Alexander Castro Campos[1], Germán Téllez Castillo[1], Francisco Javier Zaragoza Martínez[2]

[1]Laboratorio de Simulación y Modelado, Centro de Investigación en Computación
Mexico City, Mexico
acastroa09@sagitario.cic.ipn.mx, gtellez@cic.ipn.mx

[2] Departamento de Sistemas, UAM Azcapotzalco
Mexico City, Mexico
franz@correo.azc.uam.mx
*(Paper received on November 28, 2010, accepted on January 28, 2011)*

**Abstract.** Current object-oriented programming languages make a distinction between a function tied to a type (member function) and ordinary functions. Since calling member functions requires a different syntax, this lack of uniformity complicates the implementation of generic algorithms. Additionally, while non-member functions may be declared by any user, member functions can be declared only by the author of the type limiting its adaptation to future functional or syntactic requirements. Namespaces were designed to allow independent development of software components. However, invocation of some non-member functions does not work as intended with the prefix qualification imposed by the use of namespaces. In the C++ programming language the lookup rules were thought to allow convenient use of some non-member functions without namespace qualification. However the rules tend to find unwanted names and are considered dangerous for library development. We present a novel set of lookup rules that solves the problem in C++ and similar languages.

**Keywords:** lookup, namespaces, ADL, library development, software engineering.

## 1 Introduction

The generic programming paradigm tries to express algorithms with minimal assumptions about data abstractions [1]. Generic functions are usually expressed in terms of types that are not known until the function is actually called. This allows the algorithm to be used with a variety of types not known beforehand by the original author of the algorithm. However, some assumptions must be made; for example, a sorting algorithm may assume operator < is defined so it can be used to compare the elements of the sequence to be sorted.

The assumptions made by the author of a generic algorithm are classified in three groups: syntactic, semantic, and complexity assumptions [2]. The last two are usually

(C) C. Zepeda, R. Marcial, A. Sánchez
J. L. Zechinelli and M. Osorio (Eds)
Advances in Computer Science and Applications
Research in Computing Science 53, 2011, pp. 3-11

expressed in documentation (manuals, comments in the source code) and failure to meet them results in a wrong behavior of the generic algorithm. However, failure to meet syntactic assumptions may prevent the compilation and use of the generic algorithm even if just a minor adjustment is necessary. When making that adjustment is not possible, the only solution available is to rewrite the generic algorithm.

This is complicated by many programming languages with support for the object-oriented programming paradigm when they include the notion of member functions. In these languages, a member function is a function that can be declared only inside the declaration of the type to which it is tied to and it must be called with a special syntax. Consider the following example in C++ [3]:

```
struct test {
    void member_function( );
};

void non_member_function(test);

test t;
t.member_function( );
non_member_function(t);
```

Given this difference, the writer of a generic algorithm is left to choose which syntactic notation (calls to member or non-member functions) to use in the implementation of the algorithm. Unfortunately, if the former is preferred the algorithm will be unusable for types for which their original implementors did not provide the required functions.

Very few languages try to lessen this problem. For example, the C# programming language [4] allows to declare functions that share the same syntactic properties of member functions outside the declaration of a type. Unfortunately this is not sufficient: many languages (including C++, C#, Java [5], Python [6] and D [7]) restrict the user from declaring constructors (a special kind of member function) and some or all of the overloadable operators as non-member functions.

The problem becomes more noticeable when dealing with operators. Many languages provide a feature to ease the development of independent libraries, usually by allowing the grouping of names inside namespaces. To use a name declared inside a namespace from outside, it is necessary to qualify the desired name with the name of its namespace. Unfortunately namespace qualification defeats the convenience of notation and use provided by operators.

```
namespace ns {
    void f(test);
    void operator+(test);
}

+t;  // error, operator+ not in global scope

ns::f(t);          // qualification required
ns::operator+(t);  // qualification required
```

While it may be argued that operators deserve special treatment or that they could be declared in the global scope, many functions are so common that the lack of an operator to denote them in source code is purely accidental (for example, the square

root function has a well-known mathematical notation but its typing in current keyboards is not easy) and declaring them in the global scope brings back the problem of name collisions namespaces were supposed to solve.

This is also problematic for generic programming: since the actual types used are not known while implementing the algorithm, the namespaces where they (and possibly their non-member functions) are declared are also unknown. This could be partially solved by providing a way to find the namespace of a type and using explicit qualification with such namespace but this not guaranteed to work correctly every time (for some types their non-member functions could be declared in the same namespace but any author may decide to declare them in the global scope). If this path is chosen, it would be necessary to provide a compile-time reflection library to query the information about namespaces and declarations. We consider that the implementation of generic algorithms would become too cumbersome to be practical or intuitive.

## 2 The C++98 Solution

During the standarization of C++ the problem caused by the interaction of namespaces and operators was discovered [8]. To overcome this, the argument dependent lookup algorithm (ADL) was proposed and later accepted as part of the C++98 standard [9]. ADL is performed for unqualified function calls and, in general, it considers all the functions that are declared in the namespaces where the types of the arguments used in the invocation are declared, plus the functions found by the ordinary lookup. This allows convenient use of operators and other common functions:

```
namespace ns {
    struct array;

    int size(array);
    bool operator==(array, array);
}

ns::array a1;
ns::array a2;

size(a1);          // calls ns::size
a1 == a2;          // calls ns::operator==
```

Unfortunately it was not until several years later that some problems resulting from the interaction of ADL and function templates (the feature used to implement generic algorithms in C++) were identified. In C++, a function template allows the user to write algorithms for types that will be determined until it is used (template instantiation). The following example is a typical implementation of the algorithm that finds the minimum of two values:

```
template<typename T>
T min(T a, T b) {
    return (b < a ? b : a);
}

min(1, 2);      // T deduced as int
```

Unfortunately the names injected by ADL are considered equally important than those found by the ordinary lookup. This is particularly dangerous in the context of unknown types and calls to function templates since ADL may misguide the overload resolution algorithm to select overloads not intended by the implementor of a library:

```
namespace vendor {
    template<typename T>
    void g(T);

    template<typename T>
    void h(T v) {
        g(v);    // wants to call vendor::g
    }
}

struct mine;
void g(mine);    // unrelated to vendor::g

mine m;
vendor::h(m);    // global g called due to ADL
```

Library implementors must protect themselves from ADL by using namespace qualification even inside their own namespaces. This gives little to no advantage in comparison to the identifier prefix alternative used in languages with no support for namespaces (for example, C).

## 3 Subsequent Attempts for Solution

Designers of new programming languages have tried to avoid coming up with something similar to ADL in an attempt to prevent the problems present in C++. However their designs cannot handle some simple cases that ADL can. The D programming language still needs to rely on member functions and does not not provide a feature similar to the one found in C#. In the Clay programming language [10] this rough translation to C++ is valid:

```
namespace ns {
    struct s;
    void f(s);
}

ns::s v;
f(v);             // calls ns::f
```

However the call will fail if the user overloads the function, even if it could not be called:

```
void f(int);       // hides ns::f
f(v);              // error, ns::f is hidden
```

In the context of C++ a good analysis and some proposed solutions are presented in [11, 12] but in our opinion, the best attempt to fix this problem was proposed by Herb

Sutter in [13] during the works for the new C++ standard, dubbed as C++0x. The proposed change to the ADL algorithm consists on requiring the injected functions to have a parameter of the same type of the argument that causes the injection (ignoring pointer, reference, const, and array modifiers) and in the same position. This change would greatly reduce the number of functions injected by ADL (that currently includes unconstrained function templates, likely to be semantically unrelated), removing most (but not all) sources of surprise. However, what we consider a major issue remains unsolved. Consider the following namespaces:

```
namespace lib1 {
    struct s;
    void f(s);
}

namespace lib2 {
    template<typename T>
    void g(T v) {
        f(v);     // wants to call f via ADL
    }
}
```

Unfortunately, if the user declares a global variable named f, he will inadvertently interfere with lib2:

```
int f;

void g( ) {
    lib1::s v;
    lib2::g(v);   // will find the global f
}
```

It is vital to solve this problem if non-member functions are to become a viable alternative to member functions for expressing functionality of types in languages with features similar to those present in C++. The name lookup problem for non-member functions in presence of templates and namespaces is considered open by Alexander Stepanov who, along with David R. Musser, defined generic programming back in 1987 [14, 15] and are authors of the Standard Template Library [16].

## 4 Proposed Solution

A key insight for solving this problem is recognizing that ADL ignores the information about the namespace hierarchy as it unilaterally injects a set of functions that are equally important as any others, even those that were in the same scope as the invocation (the ones the user probably intended to call). We propose to apply the following changes to the lookup rules:

*1. Strengthen the requirements for ADL*

We propose a similar change to the one suggested by Sutter. However, ADL currently searches for functions in the namespaces of the arguments of struct templates and to achieve uniformity the pointer, reference, const, and array type modifiers must be considered template-like type generators:

```
template<typename T>
struct pair {
    T first, second;
};

namespace ns {
    struct inner;

    template<typename T>
    void f(T);
}

pair<ns::inner> p;

f(p);           // calls ns::f via ADL
f(&p.first);    // calls ns::f via ADL
```

We propose to drop this feature completely and require an exact match between the type of the parameter and the type of the argument, ignoring only reference and const modifiers.

*2. Continue the search after finding non-candidates*

In C++ the search for a function stops after finding one with the desired name, even if it cannot be called. This means the following will not compile:

```
void f( );

namespace ns {
    void f(int);

    void g( ) {
        f( );   // error, f(int) unusable
    }
}
```

The rationale for this rule is documented in [17] and was considered correct in the context of object oriented type hierarchies and the absence of function overloading. The introduction of overloading in C++ casted doubts about the usefulness of this rule but backward compatibility was considered important enough to keep it. We propose to remove this rule and to allow the lookup algorithm to continue searching in enclosing scopes as long as no callable function has been found. This is necessary for applying the third proposed change.

*3. Make ADL consider the namespace hierarchy*

The set of functions considered by the ADL algorithm is no longer injected as is. The functions are injected from the scope where they are declared into the enclosing scopes until the global scope is reached. We call this process *downward function propagation*.

```
namespace ns {
    struct s;
    void f(s);
}

void g( ) {
    ns::s v;
    f(v);       // ns::f found by propagation
}

namespace mine {
    void f(ns::s);

    void g( ) {
        ns::s v;
        f(v);   // mine::f found
    }
}
```

This effectively turns namespaces into semantic spaces since it is possible to hide any function, even those honored by ADL. This could prove useful for redefining builtin operations if that were to be allowed:

```
namespace safe {
    int& operator*(int* pointer) {
        if (pointer == NULL) {
            exit("null pointer dereference");
        }

        return builtin::operator*(pointer);
    }

    void f(int* p) {
        *p = 1;   // safe::operator* called
    }
}
```

Except for namespaces declared directly in the global scope, nested namespaces are typically coded by the same team of developers. Since function propagation is only performed from nested to enclosing scopes, the propagation of any unwanted names coming from nested scopes is the developer's fault. The propagation remains active only during the resolution of the function call that triggered it.

*4. Give preference to declarations found via ADL*

The propagated functions found in a given scope have preference over any other declaration:

```
namespace lib {
    struct s;
    void f(s);
 }

int f;

void g( ) {
    lib::s v;
    f(v);   // will always find lib::f
}
```

This guarantees that unknown names will not interfere if the propagation mechanism is sufficient to resolve function calls between two different scopes or namespaces. For namespaces that represent libraries, this allows library developers to safely use declarations contained in another library, isolating them from declarations that may be added in the global scope unless no propagated functions are found and the searched name cannot be resolved locally.

While not always necessary in C++ since user declarations are typically processed after library inclusions, this rule is particularly important for templates as lookup may be deferred for some names until instantiation time and in languages where lookup is not affected by the order of declarations.

## 5 Conclusions

We presented a novel set of lookup rules that solves an open problem in programming languages design by allowing practical use of non-member functions for declaring the associated functionality of types. This reduces the syntactic variation of programming languages which is particularly important for generic programming, while performing an intuitive name lookup that considers the potential semantic differences between scopes. The lookup rules do not protect the developers from themselves but protect them from declarations contained in other scopes, including the global scope if necessary. These rules were designed as part of the design of a new programming language developed in [18].

## References

1. Garcia, R., Jarvi, J., Lumnsdaine, A., Siek, J.G., Willcock, J.: A comparative study of language support for generic programming. Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications. ACM SIGPLAN Notices (2003).
2. Stepanov, A., McJones, P.: Elements of Programming. Addison-Wesley Professional (2009).
3. Stroustrup, B.: The C++ Programming Language. Addison Wesley (2000).
4. C# Language Reference. Microsoft Corporation (2007).
5. Java Language Specification. Sun Microsystems (2005).
6. The Python Language Reference, http://docs.python.org/reference/.

7. D Programming Language 2.0, http://www.digitalmars.com/d/2.0/index.html.
8. Koenig, A.: Reconciling overloaded operators with namespaces, (1995).
9. Programming languages - C++. ISO/IEC 14882 (1998).
10. Clay Programming Language, http://tachyon.in/clay/.
11. Dimov, P.: User-supplied specializations of standard library algorithms, (2001).
12. Abrahams, D.: Explicit namespaces, (2004).
13. Sutter, H.: A modest proposal: fixing ADL (revision 2), (2006).
14. Musser, D.R., Stepanov, A.: Generic Programming. Presented at the First International Joint Conference of ISSAC and AAECC, Roma, Italia (1988).
15. Stepanov, A.: Notes on Programming, (2006).
16. Musser, D.R., Derge, G.J., Saini, A.: Foreword. STL Tutorial and Reference Guide. Addison-Wesley, Boston, MA (2001).
17. Stroustrup, B.: The Design and Evolution of C++. Addison Wesley, Reading, MA (1994).
18. Castro Campos, R.A.: Un modelo de intérprete para un lenguaje de programación de sistemas basado en C, (2010).