Argument-dependent lookup

Argument-dependent lookup, also known as ADL, or Koenig lookup [1], is the set of rules for looking up the unqualified function names in function-call expressions, including implicit function calls to overloaded operators. These function names are looked up in the namespaces of their arguments in addition to the scopes and namespaces considered by the usual unqualified name lookup.

Argument-dependent lookup makes it possible to use operators defined in a different namespace. Example:

Run this code

```
#include <iostream>

int main()
{
    std::cout << "Test\n"; // There is no operator<< in global namespace, but ADL
                           // examines std namespace because the left argument is in
                           // std and finds std::operator<<(std::ostream&, const char*)
    operator<<(std::cout, "Test\n"); // same, using function call notation

    // however,
    std::cout << endl; // Error: 'endl' is not declared in this namespace.
                       // This is not a function call to endl(), so ADL does not apply

    endl(std::cout); // OK: this is a function call: ADL examines std namespace
                     // because the argument of endl is in std, and finds std::endl

    (endl)(std::cout); // Error: 'endl' is not declared in this namespace.
                       // The sub-expression (endl) is not an unqualified-id
}
```

Details

First, the argument-dependent lookup is not considered if the lookup set produced by usual unqualified lookup contains any of the following:

1) a declaration of a class member
2) a declaration of a function at block scope (that's not a using-declaration)
3) any declaration that is not a function or a function template (e.g. a function object or another variable whose name conflicts with the name of the function that's being looked up)

Otherwise, for every argument in a function call expression its type is examined to determine the *associated set of namespaces and classes* that it will add to the lookup.

1) For arguments of fundamental type, the associated set of namespaces and classes is empty.
2) For arguments of class type (including union), the set consists of
   a) The class itself
   b) All of its direct and indirect base classes
   c) If the class is a member of another class, the class of which it is a member
   d) The innermost enclosing namespaces of the classes added to the set
3) For arguments whose type is a class template specialization, in addition to the class rules, the following types are examined and their associated classes and namespaces are added to the set
   a) The types of all template arguments provided for type template parameters (skipping non-type template parameters and skipping template template parameters)
   b) The namespaces in which any template template arguments are members
   c) The classes in which any template template arguments are members (if they happen to be class member templates)
4) For arguments of enumeration type, the innermost enclosing namespace of the declaration of the enumeration type is defined is added to the set. If the enumeration type is a member of a class, that class is added to the set.
5) For arguments of type pointer to T or pointer to an array of T, the type T is examined and its associated set of classes and namespaces is added to the set.
6) For arguments of function type, the function parameter types and the function return type are examined and their associated set of classes and namespaces are added to the set.
7) For arguments of type pointer to member function F of class X, the function parameter types, the function return type, and the class X are examined and their associated set of classes and namespaces are added to the set.
8) For arguments of type pointer to data member T of class X, the member type and the type X are both examined and their associated set of classes and namespaces are added to the set.
9) If the argument is the name or the address-of expression for a set of overloaded functions (or function templates), every function in the overload set is examined and its associated set of classes and namespaces is added to the set.
   - Additionally, if the set of overloads is named by a template-id, all of its type template arguments and template template arguments (but not non-type template arguments) are examined and their associated set of classes and namespaces are added to the set.

If any namespace in the associated set of classes and namespaces is an inline namespace, its enclosing namespace is also added to the set. If any namespace in the associated set of classes and namespaces directly contains an inline namespace, that inline namespace is added to the set.	(since C++11)

After the associated set of classes and namespaces is determined, all declarations found in classes of this set are discarded for the purpose of further ADL processing, except namespace-scoped friend functions and function templates, as stated in point 2 below.

The set of declarations found by ordinary unqualified lookup and the set of declarations found in all elements of the associated set produced by ADL, are merged, with the following special rules

1) using-directives in the associated namespaces are ignored
2) namespace-scoped friend functions (and function templates) that are declared in an associated class are visible through ADL even if they are not visible through ordinary lookup
3) all names except for the functions and function templates are ignored (no collision with variables)

Notes

Because of argument-dependent lookup, non-member functions and non-member operators defined in the same namespace as a class are considered part of the public interface of that class (if they are found through ADL) [2].

ADL is the reason behind the established idiom for swapping two objects in generic code:

```
using std::swap;
swap(obj1, obj2);
```

because calling `std::swap(obj1, obj2)` directly would not consider the user-defined swap() functions that could be defined in the same namespace as the types of obj1 or obj2, and just calling the unqualified `swap(obj1, obj2)` would call nothing if no user-defined overload was provided. In particular, `std::iter_swap` and all other standard library algorithms use this approach when dealing with *Swappable* types.

Name lookup rules make it impractical to declare operators in global or user-defined namespace that operate on types from the std namespace, e.g. a custom `operator>>` or `operator+` for `std::vector` or for `std::pair` (unless the element types of the vector/pair are user-defined types, which would add their namespace to ADL). Such operators would not be looked up from template instantiations, such as the standard library algorithms. See dependent names for further details.

ADL can find a friend function (typically, an overloaded operator) that is defined entirely within a class or class template, even if it was never declared at namespace level.

```
template<typename T>
struct number
{
    number(int);
    friend number gcd(number x, number y) { return 0; }; // definition within
                                                         // a class template
};

// unless a matching declaration is provided gcd is
// an invisible (except through ADL) member of this namespace
void g()
{
    number<double> a(3), b(4);
    a = gcd(a, b); // finds gcd because number<double> is an associated class,
                   // making gcd visible in its namespace (global scope)
//  b = gcd(3, 4); // Error; gcd is not visible
}
```

Although a function call can be resolved through ADL even if ordinary lookup finds nothing, a function call to a function template with explicitly-specified template arguments requires that there is a declaration of the template found by ordinary lookup (otherwise, it is a syntax error to encounter an unknown name followed by a less-than character)

```
namespace N1
{
    struct S {};

    template<int X>
    void f(S);
}

namespace N2
{
    template<class T>
    void f(T t);
}

void g(N1::S s)
{
    f<3>(s);     // Syntax error until C++20 (unqualified lookup finds no f)
    N1::f<3>(s); // OK, qualified lookup finds the template 'f'
    N2::f<3>(s); // Error: N2::f does not take a non-type parameter
                 //        N1::f is not looked up because ADL only works
                 //              with unqualified names

    using N2::f;
    f<3>(s); // OK: Unqualified lookup now finds N2::f
             //     then ADL kicks in because this name is unqualified
             //     and finds N1::f
}
```

(until C++20)

In the following contexts ADL-only lookup (that is, lookup in associated namespaces only) takes place:

- the lookup of non-member functions begin and end performed by the range-for loop if member lookup fails (since C++11)
- the dependent name lookup from the point of template instantiation.
- the lookup of non-member function get performed by structured binding declaration for tuple-like types (since C++17)

Examples

This section is incomplete
Reason: more examples

Example from http://www.gotw.ca/gotw/030.htm

Run this code

```
namespace A
{
    struct X;
    struct Y;

    void f(int);
    void g(X);
}

namespace B
{
    void f(int i)
    {
        f(i); // calls B::f (endless recursion)
    }

    void g(A::X x)
    {
        g(x); // Error: ambiguous between B::g (ordinary lookup)
              //        and A::g (argument-dependent lookup)
    }

    void h(A::Y y)
    {
        h(y); // calls B::h (endless recursion): ADL examines the A namespace
              // but finds no A::h, so only B::h from ordinary lookup is used
    }
}
```

Defect reports

The following behavior-changing defect reports were applied retroactively to previously published C++ standards.

DR	Applied to	Behavior as published	Correct behavior
CWG 33 (https://cplusplus.github.io/CWG/issues/33.html)	C++98	the associated namespaces or classes are unspecified if an argument used for lookup is the address of a group of overloaded functions or a function template	specified
CWG 90 (https://cplusplus.github.io/CWG/issues/90.html)	C++98	the associated classes of a nested non-union class did not include its enclosing class, but a nested union was associated with its enclosing class	non-unions also associated
CWG 239 (https://cplusplus.github.io/CWG/issues/239.html)	C++98	a block-scope function declaration found in the ordinary	ADL not considered except

		unqualified lookup did not prevent ADL from happening	for using-declarations
CWG 997 (https://cplusplus.github.io/CWG/issues/997.html)	C++98	dependent parameter types and return types were excluded from consideration in determining the associated classes and namespaces of a function template	included
CWG 1690 (https://cplusplus.github.io/CWG/issues/1690.html)	C++98 C++11	ADL could not find lambdas (C++11) or objects of local class types (C++98) that are returned	they can be found
CWG 1691 (https://cplusplus.github.io/CWG/issues/1691.html)	C++11	ADL had surprising behaviors for opaque enumeration declarations	fixed
CWG 1692 (https://cplusplus.github.io/CWG/issues/1692.html)	C++98	doubly-nested classes did not have associated namespaces (their enclosing classes are not members of any namespace)	associated namespaces are extended to the innermost enclosing namespaces

See also

- Name lookup
- Template argument deduction
- Overload resolution

External links

1. ↑ Andrew Koenig: "A Personal Note About Argument-Dependent Lookup" (https://www.drdobbs.com/cpp/a-personal-note-about-argument-dependent/232901443)
2. ↑ H. Sutter (1998) "What's In a Class? - The Interface Principle" (http://www.gotw.ca/publications/mill02.htm) in C++ Report, 10(3)