Unqualified name lookup

For an *unqualified* name, that is a name that does not appear to the right of a scope resolution operator `::`, name lookup examines the scopes as described below, until it finds at least one declaration of any kind, at which time the lookup stops and no further scopes are examined. (Note: lookup from some contexts skips some declarations, for example, lookup of the name used to the left of `::` ignores function, variable, and enumerator declarations, lookup of a name used as a base class specifier ignores all non-type declarations)

For the purpose of unqualified name lookup, all declarations from a namespace nominated by a using directive appear as if declared in the nearest enclosing namespace which contains, directly or indirectly, both the using-directive and the nominated namespace.

Unqualified name lookup of the name used to the left of the function-call operator (and, equivalently, operator in an expression) is described in argument-dependent lookup.

File scope

For a name used in global (top-level namespace) scope, outside of any function, class, or user-declared namespace, the global scope before the use of the name is examined:

```
int n = 1;     // declaration of n
int x = n + 1; // OK: lookup finds ::n

int z = y - 1; // Error: lookup fails
int y = 2;     // declaration of y
```

Namespace scope

For a name used in a user-declared namespace outside of any function or class, this namespace is searched before the use of the name, then the namespace enclosing this namespace before the declaration of this namespace, etc until the global namespace is reached.

```
int n = 1; // declaration

namespace N
{
    int m = 2;

    namespace Y
    {
        int x = n; // OK, lookup finds ::n
        int y = m; // OK, lookup finds ::N::m
        int z = k; // Error: lookup fails
    }

    int k = 3;
}
```

Definition outside of its namespace

For a name used in the definition of a namespace-member variable outside the namespace, lookup proceeds the same way as for a name used inside the namespace:

```
namespace X
{
    extern int x; // declaration, not definition
    int n = 1;    // found 1st
}

int n = 2;        // found 2nd
int X::x = n;     // finds X::n, sets X::x to 1
```

Non-member function definition

For a name used in the definition of a function, either in its body or as part of default argument, where the function is a member of user-declared or global namespace, the block in which the name is used is searched before the use of the name, then the enclosing block is searched before the start of that block, etc, until reaching the block that is the function body. Then the namespace in which the function is declared is searched until the definition (not necessarily the declaration) of the function that uses the name, then the enclosing namespaces, etc.

```
namespace A
{
    namespace N
    {
        void f();
        int i = 3; // found 3rd (if 2nd is not present)
    }

    int i = 4;     // found 4th (if 3rd is not present)
}

int i = 5;         // found 5th (if 4th is not present)

void A::N::f()
{
    int i = 2;     // found 2nd (if 1st is not present)

    while (true)
    {
       int i = 1;  // found 1st: lookup is done
       std::cout << i;
    }
}

// int i;          // not found

namespace A
{
    namespace N
    {
        // int i;  // not found
    }
}
```

Class definition

For a name used anywhere in class definition (including base class specifiers and nested class definitions), except inside a member function body, a default argument of a member function, exception specification of a member function, or default member initializer, where the member may belong to a nested class whose definition is in the body of the enclosing class, the following scopes are searched:

a) the body of the class in which the name is used until the point of use,
b) the entire body of its base class(es), recursing into their bases when no declarations are found,
c) if this class is nested, the body of the enclosing class until the definition of this class and the entire body of the base class(es) of the enclosing class,
d) if this class is local, or nested within a local class, the block scope in which the class is defined until the point of definition,
e) if this class is a member of a namespace, or is nested in a class that is a member of a namespace, or is a local class in a function that is a member of a namespace, the scope of the namespace is searched until the definition of the class, enclosing class, or function; lookup continues to the namespaces enclosing that one until the global scope.

For a friend declaration, the lookup to determine whether it refers to a previously declared entity proceeds as above except that it stops after the innermost enclosing namespace.

```
namespace M
{
    // const int i = 1; // never found

    class B
    {
        // static const int i = 3;     // found 3rd (but will not pass access check)
    };
}

// const int i = 5;                    // found 5th

namespace N
{
    // const int i = 4;                // found 4th

    class Y : public M::B
    {
        // static const int i = 2;     // found 2nd

        class X
        {
            // static const int i = 1; // found 1st
            int a[i]; // use of i
            // static const int i = 1; // never found
        };

        // static const int i = 2;     // never found
    };

    // const int i = 4;                // never found
}

// const int i = 5;                    // never found
```

Injected class name

For the name of a class or class template used within the definition of that class or template or derived from one, unqualified name lookup finds the class that's being defined as if the name was introduced by a member declaration (with public member access). For more detail, see injected-class-name.

Member function definition

For a name used inside a member function body, a default argument of a member function, exception specification of a member function, or a default member initializer, the scopes searched are the same as in class definition, except that the entire scope of the class is considered, not just the part prior to the declaration that uses the name. For nested classes the entire body of the enclosing class is searched.

```
class B
{
    // int i;         // found 3rd
};

namespace M
{
    // int i;         // found 5th

    namespace N
    {
        // int i;     // found 4th

        class X : public B
        {
            // int i; // found 2nd
            void f();
            // int i; // found 2nd as well
        };

        // int i;     // found 4th
    }
}

// int i;             // found 6th

void M::N::X::f()
{
    // int i;         // found 1st
    i = 16;
    // int i;         // never found
}

namespace M
{
    namespace N
    {
        // int i;     // never found
    }
}
```

Either way, when examining the bases from which the class is derived, the following rules, sometime referred to as dominance in virtual inheritance , are followed:

A member name found in a sub-object `B` hides the same member name in any sub-object `A` if `A` is a base class sub-object of `B`. (Note that this does not hide the name in any additional, non-virtual, copies of `A` on the inheritance lattice that aren't bases of `B`: this rule only has an effect on virtual inheritance.) Names introduced by using-declarations are treated as names in the class containing the declaration. After examining each base, the resulting set must either include declarations of a static member from subobjects of the same type, or declarations of non-static members from the same subobject. (until C++11)
A *lookup set* is constructed, which consists of the declarations and the subobjects in which these declarations were found. Using-declarations are replaced by the members they represent and type declarations, including injected-class-names are replaced by the types they represent. If `C` is the class in whose scope the name was used, `C` is examined first. If the list of declarations in `C` is empty, lookup set is built for each of its direct bases `Bi` (recursively applying these rules if `Bi` has its own bases). Once built, the lookup sets for the direct bases are merged into the lookup set in `C` as follows: ▪ if the set of declarations in `Bi` is empty, it is discarded, ▪ if the lookup set of `C` built so far is empty, it is replaced by the lookup set of `Bi`, ▪ if every subobject in the lookup set of `Bi` is a base of at least one of the subobjects already added to the lookup set of `C`, the lookup set of `Bi` is discarded, (since C++11) ▪ if every subobject already added to the lookup set of `C` is a base of at least one subobject in the lookup set of `Bi`, then the lookup set of `C` is discarded and replaced with the lookup set of `Bi`, ▪ otherwise, if the declaration sets in `Bi` and in `C` are different, the result is an ambiguous merge: the new lookup set of `C` has an invalid declaration and a union of the subobjects earlier merged into `C` and introduced from `Bi`. This invalid lookup set may not be an error if it is discarded later, ▪ otherwise, the new lookup set of `C` has the shared declaration sets and the union of the subobjects earlier merged into `C` and introduced from `Bi`.

```
struct X { void f(); };

struct B1: virtual X { void f(); };

struct B2: virtual X {};

struct D : B1, B2
{
    void foo()
    {
        X::f(); // OK, calls X::f (qualified lookup)
        f(); // OK, calls B1::f (unqualified lookup)
    }
};

// C++98 rules: B1::f hides X::f, so even though X::f can be reached from D
// through B2, it is not found by name lookup from D.

// C++11 rules: lookup set for f in D finds nothing, proceeds to bases
//  lookup set for f in B1 finds B1::f, and is completed
// merge replaces the empty set, now lookup set for f in C has B1::f in B1
//  lookup set for f in B2 finds nothing, proceeds to bases
//    lookup for f in X finds X::f
//  merge replaces the empty set, now lookup set for f in B2 has X::f in X
// merge into C finds that every subobject (X) in the lookup set in B2 is a base
// of every subobject (B1) already merged, so the B2 set is discarded
// C is left with just B1::f found in B1
// (if struct D : B2, B1 was used, then the last merge would *replace* C's
//  so far merged X::f in X because every subobject already added to C (that is X)
//  would be a base of at least one subobject in the new set (B1), the end
//  result would be the same: lookup set in C holds just B1::f found in B1)
```

Unqualified name lookup that finds static members of `B`, nested types of `B`, and enumerators declared in `B` is unambiguous even if there are multiple non-virtual base subobjects of type `B` in the inheritance tree of the class being examined:

```
struct V { int v; };

struct A
{
    int a;
    static int s;
    enum { e };
};

struct B : A, virtual V {};
struct C : A, virtual V {};
struct D : B, C {};

void f(D& pd)
{
    ++pd.v;       // OK: only one v because only one virtual base subobject
    ++pd.s;       // OK: only one static A::s, even though found in B and in C
    int i = pd.e; // OK: only one enumerator A::e, even though found in B and C
    ++pd.a;       // error, ambiguous: A::a in B and A::a in C
}
```

Friend function definition

For a name used in a friend function definition inside the body of the class that is granting friendship, unqualified name lookup proceeds the same way as for a member function. For a name used in a friend function which is defined outside the body of a class, unqualified name lookup proceeds the same way as for a function in a namespace.

```
int i = 3;                    // found 3rd for f1, found 2nd for f2

struct X
{
    static const int i = 2;    // found 2nd for f1, never found for f2

    friend void f1(int x)
    {
        // int i;              // found 1st
        i = x;                 // finds and modifies X::i
    }

    friend int f2();

    // static const int i = 2; // found 2nd for f1 anywhere in class scope
};

void f2(int x)
{
    // int i;                  // found 1st
```

```
    i = x;                      // finds and modifies ::i
}
```

Friend function declaration

For a name used in the declarator of a friend function declaration that friends a member function from another class, if the name is not a part of any template argument in the declarator identifier, the unqualified lookup first examines the entire scope of the member function's class. If not found in that scope (or if the name is a part of a template argument in the declarator identifier), the lookup continues as if for a member function of the class that is granting friendship.

```
template<class T>
struct S;

// the class whose member functions are friended
struct A
{
    typedef int AT;

    void f1(AT);
    void f2(float);

    template<class T>
    void f3();

    void f4(S<AT>);
};

// the class that is granting friendship for f1, f2 and f3
struct B
{
    typedef char AT;
    typedef float BT;

    friend void A::f1(AT);    // lookup for AT finds A::AT (AT found in A)
    friend void A::f2(BT);    // lookup for BT finds B::BT (BT not found in A)
    friend void A::f3<AT>();  // lookup for AT finds B::AT (no lookup in A, because
                              //     AT is in the declarator identifier A::f3<AT>)
};

// the class template that is granting friendship for f4
template<class AT>
struct C
{
    friend void A::f4(S<AT>); // lookup for AT finds A::AT
                              // (AT is not in the declarator identifier A::f4)
};
```

Default argument

For a name used in a default argument in a function declaration, or name used in the *expression* part of a member-initializer of a constructor, the function parameter names are found first, before the enclosing block, class, or namespace scopes are examined:

```
class X
{
    int a, b, i, j;
public:
    const int& r;

    X(int i): r(a),      // initializes X::r to refer to X::a
              b(i),      // initializes X::b to the value of the parameter i
              i(i),      // initializes X::i to the value of the parameter i
              j(this->i) // initializes X::j to the value of X::i
    {}
};

int a;
int f(int a, int b = a); // error: lookup for a finds the parameter a, not ::a
                         // and parameters are not allowed as default arguments
```

Static data member definition

For a name used in the definition of a static data member, lookup proceeds the same way as for a name used in the definition of a member function.

```
struct X
{
    static int x;
    static const int n = 1; // found 1st
};

int n = 2;                  // found 2nd
int X::x = n;               // finds X::n, sets X::x to 1, not 2
```

Enumerator declaration

For a name used in the initializer part of the enumerator declaration, previously declared enumerators in the same enumeration are found first, before the unqualified name lookup proceeds to examine the enclosing block, class, or namespace scope.

```
const int RED = 7;

enum class color
{
    RED,
    GREEN = RED + 2, // RED finds color::RED, not ::RED, so GREEN = 2
    BLUE = ::RED + 4 // qualified lookup finds ::RED, BLUE = 11
};
```

Catch clause of a function-try block

For a name used in the catch-clause of a function-try-block, lookup proceeds as if for a name used in the very beginning of the outermost block of the function body (in particular, function parameters are visible, but names declared in that outermost block are not)

```
int n = 3;          // found 3rd
int f(int n = 2)    // found 2nd

try
{
    int n = -1;     // never found
}
catch(...)
{
    // int n = 1;   // found 1st
    assert(n == 2); // loookup for n finds function parameter f
    throw;
}
```

Overloaded operator

For an operator used in expression (e.g., `operator+` used in a+b), the lookup rules are slightly different from the operator used in an explicit function-call expression such as `operator+(a,b)`: when parsing an expression, two separate lookups are performed: for the non-member operator overloads and for the member operator overloads (for the operators where both forms are permitted). Those sets are then merged with the built-in operator overloads on equal grounds as described in overload resolution. If explicit function call syntax is used, regular unqualified name lookup is performed:

```
struct A {};
void operator+(A, A);  // user-defined non-member operator+

struct B
{
    void operator+(B); // user-defined member operator+
    void f();
};

A a;

void B::f() // definition of a member function of B
{
    operator+(a, a); // error: regular name lookup from a member function
                     // finds the declaration of operator+ in the scope of B
                     // and stops there, never reaching the global scope

    a + a; // OK: member lookup finds B::operator+, non-member lookup
           // finds ::operator+(A,A), overload resolution selects ::operator+(A,A)
}
```

Template definition

For a non-dependent name used in a template definition, unqualified name lookup takes place when the template definition is examined. The binding to the declarations made at that point is not affected by declarations visible at the point of instantiation. For a dependent name used in a template definition, the lookup is postponed until the template arguments are known, at which time ADL examines function declarations with external linkage (until C++11) that are visible from the template definition context as well as in the template instantiation context, while non-ADL lookup only examines function declarations with external linkage (until C++11) that are visible from the template definition context (in other words, adding a new function declaration after template definition does not make it visible except via ADL). The behavior is undefined if there is a better match with external linkage in the namespaces examined by the ADL lookup, declared in some other translation unit, or if the lookup would have been ambiguous if those translation units were examined. In any case, if a base class depends on a template parameter, its scope is not examined by unqualified name lookup (neither at the point of definition nor at the point of instantiation).

```
void f(char); // first declaration of f

template<class T>
void g(T t)
{
    f(1);    // non-dependent name: lookup finds ::f(char) and binds it now
    f(T(1)); // dependent name: lookup postponed
    f(t);    // dependent name: lookup postponed
//  dd++;    // non-dependent name: lookup finds no declaration
}

enum E { e };
void f(E);   // second declaration of f
void f(int); // third declaration of f
double dd;

void h()
{
    g(e);  // instantiates g<E>, at which point
           // the second and the third uses of the name 'f'
           // are looked up and find ::f(char) (by lookup) and ::f(E) (by ADL)
           // then overload resolution chooses ::f(E).
           // This calls f(char), then f(E) twice

    g(32); // instantiates g<int>, at which point
           // the second and the third uses of the name 'f'
           // are looked up and find ::f(char) only
           // then overload resolution chooses ::f(char)
           // This calls f(char) three times
}

typedef double A;

template<class T>
class B
{
    typedef int A;
};

template<class T>
struct X : B<T>
{
    A a; // lookup for A finds ::A (double), not B<T>::A
};
```

Note: see dependent name lookup rules for the reasoning and implications of this rule.

Template name

This section is incomplete
Reason: dual-scope lookup of the template name after -> and .

Member of a class template outside of template

This section is incomplete

References

- C++23 standard (ISO/IEC 14882:2023):
  - 6.5 Name lookup [basic.lookup] (p: 44-45)
  - 6.5.2 Member name lookup [class.member.lookup] (p: 45-47)
  - 13.8 Name resolution [temp.res] (p: 399-403)
- C++20 standard (ISO/IEC 14882:2020):
  - 6.5 Name lookup [basic.lookup] (p: 38-50)
  - 11.8 Member name lookup [class.member.lookup] (p: 283-285)
  - 13.8 Name resolution [temp.res] (p: 385-400)
- C++17 standard (ISO/IEC 14882:2017):
  - 6.4 Name lookup [basic.lookup] (p: 50-63)
  - 13.2 Member name lookup [class.member.lookup] (p: 259-262)
  - 17.6 Name resolution [temp.res] (p: 375-378)
- C++14 standard (ISO/IEC 14882:2014):
  - 3.4 Name lookup [basic.lookup] (p: 42-56)
  - 10.2 Member name lookup [class.member.lookup] (p: 233-236)
  - 14.6 Name resolution [temp.res] (p: 346-359)
- C++11 standard (ISO/IEC 14882:2011):
  - 3.4 Name lookup [basic.lookup]
  - 10.2 Member name lookup [class.member.lookup]
  - 14.6 Name resolution [temp.res]
- C++98 standard (ISO/IEC 14882:1998):
  - 3.4 Name lookup [basic.lookup]
  - 10.2 Member name lookup [class.member.lookup]
  - 14.6 Name resolution [temp.res]

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 490 (https://cplusplus.github.io/CWG/issues/490.html)	C++98	any name in a template argument in a friend member function declaration was not looked up in the scope of the member function's class	only excludes the names in template arguments in the declarator identifier
CWG 514 (https://cplusplus.github.io/CWG/issues/514.html)	C++98	any unqualified name used in namespace scope was first looked up in that scope	the unqualified names used to define a namespace variable member outside that namespace are first looked up in that namespace

See also

- Qualified name lookup
- Scope
- Argument-dependent lookup
- Template argument deduction
- Overload resolution