Document Number: X3J16/92-0124
WG21/N0201
Date: November 5, 1992
Revised: January 20, 1993
Project: Programming Language C++
Reply to: Tom Pennello
tom@metaware.com

How to evaluate class-name after . and ->

This proposal discusses the evaluation of id-expression in the expression

```
postfix-expression . id-expression
postfix-expression -> id-expression
```

In particular, it tells how to evaluate A:: in an expression of the form x.A:: or x.A:: . It does not discuss any change in the rules for evaluating x.member.

Problem

The draft, section 5.2.4 Class Member Access, discusses expressions of the form

```
postfix-expression . id-expression
postfix-expression -> id-expression
```

and says that the id-expression must name a member of that class. We all agree that ".identifier" or "->identifier" implies looking up that identifier in the class or any of its base classes. But the draft does not address what we do with "x.A::member", i.e, where id-expression is a qualified-id. Where do we look up A? In the current scope? A is almost never a member of the class of x, so the same rule we use for "x.identifier" -- look up in the class of x -- can't work for "x.A::member". We have to consider alternative places to lookup A. Here is typical existing practice:

```
struct A { int x; };
struct B: A {};
main () {
    B b;
    b.x = 1;            // 1
    b.A::x = 1;         // 2
    }
```

Example 1.

In //1 we look up x in B (and any base classes; this is implied by the meaning of inheritance). For //2, A is not in B (or any base classes). Instead, it's found in the global scope. Where, in general, should it be looked for?

A rule is needed only for the FIRST identifier preceding the ::, because after the ::, the rules are alread spelled out in the draft. For example, in

```
A::B::C::m
```

B must be a member of (the class that) A (denotes), C a member of A::B, and m a member of A::B::C. Thus we can restrict ourselves to figuring out how to lookup A. Note that B and C are the actual names of declarations in classes, whereas A could be a typedef name referring to a class, or even an expression (such as a template instantiation) referring to a class.

Motivations

These are the properties that motivate the solution described here:

M1 that the usage of x.A:: in a template class body is explained by the rules without introducing any new rules for such an expression specifically for templates. "A" might be a template instantiation expression such as buffer<1024,char>, as in x.buffer<1024,char>::member.

M2 that the use of this->A:: and the use of A:: mean the same thing in a member function of the class of *this.

M3 Existing practice must be accommodated as much as possible.

Other motivations included

M4 Other solutions for "x.A::" have proposed "search for A as a base class name in the class of x". But the problem is that base classes in a class definition are not names; they are expressions that denote base classes. For example, "struct B : C::D, E::F, buf<char,1024> { ..." . What do we then lookup with "x.C::D::... or with x.buf<char,1024>"? Do we observe that C::D is an expression identical to a base expression?

The motivation is to prevent ever needing to have to "look for base class names".

The motivations show that I am interested in solving more than one problem in the language, but I regard that as a benefit.

We start exploring the solution by considering an example addressing motivation 2. It is common C++ practice to elide "this->" when referring to a member. The implication is that the elision produces something that is identical to the "this->" expression. Alternatively, if you have an expression that denotes a member, you can put this-> in front of it and get the same result.

For simple member names the expressions are provably identical.

```
struct B { int m; };
struct D:B {
    int m;
    f() {
        m = 1;          //1 member m
        this->m = 1;    //2 must be the same as //1
        B::m = 1;       //3 member ::B::m
        this->B::m = 1; //4 hopefully the same as //2
        }
    }
```

Example 2

Here "m" and "this->m" can be shown to be the same, assuming //1 denotes a member of the class. The reason is that the lookup of m in //1 searches the function local scopes first, and if not found, the class scope (= class+bases) are searched. The rule for //2 is to just search the class scope. So a difference arises only if //1 finds something in a local scope. If m is locally redeclared, //1 can't possibly denote a member, so a local redeclaration always makes //1 and //2 different.

We now propose a rule that makes the //3 the same as //4.

> Rule. In an expression of the form
>
> x.expn::...
>
> p->expn::...
>
> expn is evaluated in a context consisting of
>
> - first, the class scope of x (*p)
> - second, of the scope containing the expression (i.e., as if expn appeared free at the point of its writing).

Here expn can be a simple name or something complicated, such as a template instantiation.

By the way, I presume that the "class scope of x" means x's class and all its bases, then x's lexical class parent and all its bases, etc, but stopping (and not including) the first non-class scope. That's apparently the Lund lookup rule for member functions. One might think of this as the "transitive class scope of x", but we'll just call it the "class scope of x" in this paper.

The Rule makes

```
this->B::m
```

the same as

```
B::m
```

in the same situations where

```
this->x
```

is the same as

```
x
```

-- namely, where the nothing has been redeclared in the local scope that changes the meaning.

Here is an example illustrating sameness and lack of sameness:

```
struct W { static int m; };
struct B {
    int m;
    typedef W B;        //0
    };
struct Z { static int m; };
struct D:B {
    int m,n;
    f() {
        m = 1;          //1 member m
        this->m = 1;    //2 same as //1

        int n;          //3 redefine n.
        n = 1;          //4 get local n
        this->n = 1;    //5 member n, not same as //4.

        B::m = 1;       //5 member ::B::m
        this->B::m = 1; //6 the same as //5

        typedef Z B;    //7 redefine B.
        B::m = 1;       //8 get local B, and ::Z::m.
        this->B::m = 1; //9 gets ::B::B::m = ::W::m, not same as //8.
        }
```

```
        }
```

Example 2a

Example 2a shows how the same kind of redeclarations that destroy the equivalence of "m" and "this->m" also can destroy the equivalence of "B::m" and "this->B::m". But avoiding those redeclarations ensures the equivalence in both cases. Notice, however, that we had to work harder to make //9 different from //8; it took not only the redeclaration of B in //7, but also the declaration of B in //0. this->B::m finds B in the scope of *this, namely ::B::B, and so the expression refers to :B::W::m. Were we to delete //0, the B in this->B::m would not be found in the class scope of *this, so then it would be looked up as if written free, and would evaluate to the same as B::m //8. Although it is harder to make B::m and this->B::m different by redeclaration of B, it is still possible. The Rule guarantees equality only when B isn't redeclared.

As a sanity check, revisiting common practice -- viz., Example 1 -- we find:

```
struct A { int x; };
struct B: A {};
main () {
    B b;
    b.x = 1;          // 1
    b.A::x = 1;       // 2
    }
```

Example 1 (reprinted by permission)

In //2, A is not find in the class scope of B, so it is looked up "free", and ::A is found. This common practice is the primary reason that the free lookup is included in the rule. Basically, the only difference between b.x and b.A::x is that free lookup may be used for A::.

Here is a more involved example; equivalence is maintained even though a redefinition occurs.

```
struct C { struct B { int m; }; };
struct A { struct B { int m; }; };
struct D: A::B, C::B {  // Two bases of D, both nested classes named B.
    f() {
        typedef C A;     //1 Note this typedef! -- it hides ::A.
        A::B::m;         //2 Finds ::C::B::m, an inherited member of D.
        this->A::B::m;   //3 Finds ::C::B::m, the same member.
        }
    };
```

Example 3

Here the typedef in //1 changes the evaluation of A in A::B::m //2. A refers to ::C via //1; B is a member of ::C, and it contains member m, which is an inherited member of D because D derives from C::B (as its second base). In //3, A is not found in the class of *this (D), so it is again looked up free, and the local A is found. The end result is that we again find the same A::B::m, namely ::C::B::m.

This use of the typedef may be considered somewhat perverse, so here is a more reasonable example:

```
struct C { struct B { int m; }; };
```

```
struct A { struct B { int m; }; };
struct D: A::B, C::B {
    f() {
        typedef A::B firstbase; //1 Nicer name for my first base class.
        firstbase::m;           //2 Finds ::A::B::m.
        this->firstbase::m;     //3 Finds the same ::A::B::m.
        }
    };
```

Example 4

Here we have used "firstbase"//1 as a shorthand to refer to the first base class of D. The shorthand might be even more important in templates where a base can be a template expansion:

```
template <class A> class Q : other_template<A> {
    int m;
    f() {
        typedef other_template<A> mybase;
        mybase::m = 1;          //1 Refer to base's m, not Q's m.
        this->mybase::m = 1;    //2 Same here.
        // ... lots of uses of mybase ...
        }
    };
```

Example 5

Now we cannot guarantee that //2 and //1 are the same, because we don't know whether other_template<A> has the name "mybase" in it. If it does, that value will be found before the local "mybase". In general it is not possible to obtain such a guarantee. A programmer might adopt a convention whereby identifiers of certain lexical style are not used as class members, and use only those such identifiers, rather than "mybase":

```
template <class A> class Q : other_template<A> {
    int m;
    f() {
        typedef other_template<A> __non_membr;
        __non_membr::m = 1;             //1 Refer to base's m, not Q's m.
        this->__non_membr::m = 1;       //2 Same here.
        // ... lots of uses of __non_membr ...
        }
    };
```

Example 5a

Here __non_membr is not in other_template<A> due to self-imposed conventions.

The next example shows the classic "inherited" example where the base class is renamed "inherited".

```
struct B { int m; };
struct D:B {
    int m;
    typedef B inherited;
    f() {
        inherited::m = 1;       //1 member ::B::m
        this->inherited::m = 1; //2 must be the same as //1
        }
    }
```

Example 6

The Rule, although first motivated by expressions within member functions (motivation M2 above), takes care of expressions anywhere.

Consider clients of Example 6's class D:

```
typedef int inherited;  // Not found by lookups below.
func() {
    D d;
    d.inherited::m;     //1 Gets ::B::m.
    }
D glob_d;
int *p = &glob_d.inherited::m;  //2 Gets ::B::m.
```

Example 7

inherited is found in D, so we obtain ::B::m. It makes the use by the clients behave the same as if d.inherited were written in a member function of D.

Free lookup allows the same kind of functionality in client functions as we we saw in the member function of Example 4:

```
func () {
    D d;
    typedef A::B firstbase;
    d.firstbase::m = 1; //1 Gets ::A::B::m.
    }
```

Example 8

Since firstbase is not in D, it is found locally, in the function's scope.

That the class scope is looked up first prevents breaking the "inherited" paradigm in the context of two classes, both of which are using "inherited". In the following example, both classes D1 and D2 use "inherited".

```
struct B1 { int m; };
struct D1:B1 {
    typedef B1 inherited;
    f() { this->inherited::m; } //1 Use inherited within D1.
    };
struct B2 { int m; };
struct D2:B2 {
    typedef B2 inherited;
    void f() {
        D1 d1;
        d1.inherited::m;        //2 Use d1's inherited, not D2's.
        }
    };
```

Example 9

Here d1.inherited::m accesses the inherited member within D1, because inherited is found in the D1's class scope before D2's class scope is searched.

The Rule takes care of the problem of expressions within templates. It allows the use of the typedef in Example 5, but also allows expressions to denote the base. Here is a variant of Example 5:

```
template <class A> class Q : other_template<A>, A {
    int m;
    f() {
        other_template<A>::m = 1;        //1 Refer to base's m, not Q's m.
        this->other_template<A>::m = 1; //2 Same here.
        this->A::m = 1;                  //3 Some other m in A.
        }
    };
class parm { int m; };
typedef Q<parm> dummy;
```

Example 10

When Q<parm> is expanded, the formal A assumes the value parm; I tend to think of the expansion as operating by introducing typedefs to stand for the formals, as follows:

```
typedef parm A;
class Q<parm> : other_template<A>, A {
    int m;
    f() {
        other_template<A>::m = 1;        //1 Refer to base's m, not Q's m.
        this->other_template<A>::m = 1; //2 Same here.
        this->A::m = 1;                  //3 Some other m in A.
        }
    };
```

Example 10, cont'd

Here the use of A in //3 finds the A in the typedef, obtaining the proper value for the type (parm). The expression other_template<A> in //2, when evaluated, finds other_template (wherever it is) and finds the typedef A as its parameter, and is hence able to do the template expansion (this assumes that A is not defined in a base class of Q<parm>, i.e. in other_template<A> or in A). This shows again that we need not appeal to "search for class base names" as a rule; we don't have such a name in //2. We have a (type) expression. The Rule neatly takes care of these expressions.

(The introduction of typedefs to clarify template instantiation is just my interpretation of how instantiation works; the draft is wholly inadequate in describing the process of instantiation. It appears to admit either a macro-like expansion, where the same expression is repeated multiple times (which may result in repeated side effects, such as i++), or a function-call like semantics, which causes the evaluation of the template formals only once. More work needs to be done to finish off the template extension to C++.)

Here is a variant on Example 9 that shows another . expression not within a function; it's the default parameter to D2::f:

```
struct B1 { int m; };
struct D1:B1 {
    typedef B1 inherited;
    f() { this->inherited::m; } //1 Use inherited within D1.
    };
struct B2 { int m; };
D1 d1_global;
struct D2:B2 {
    typedef B2 inherited;
    void f(int def_parm = d1_global.inherited::m) {     //2
        D1 d1_local;
        d1_local.inherited::m;  //3 Use d1_local's inherited, not D2's.
        }
    };
```

Example 12

Here, in both d1_global.inherited::m//2 and d1::inherited::m//3, inherited::m accesses the inherited member within D1, because inherited is found in the D1's class scope before D2's class scope is searched.

The Rule also handles expressions of the form

```
x.::A:: ...
p->::A:: ...
```

Even though the context of the evaluation is first the class scope of x (*p), the :: before the A cause the evaluation to occur in global scope, so the class scope doesn't matter. A compiler could optimize the evaluation of these expressions by not bothering with the class scope.

Note finally that we haven't changed the rule in 5.1/8 for text preceding ::, which says "if a class-name is hidden by a nontype name in the same scope, the class-name is still found and used". For example:

```
struct A { int x; };
struct B: A {};
int A;                  // 0 hide A of struct A.
main () {
    B b;
    b.x = 1;            // 1
    b.A::x = 1;         // 2 still finds ::A::x.
    }
```

Example 1a = Example 1 modified slightly by //0.

Compiler implementation.

When a compiler sees p->A::x, it can take the following actions:

```
open a local scope
enter in this scope all the names in the class scope of *p
evaluate A
close the local scope
```

This is a bit more involved than a possible alternative:

```
lookup A in the scope of *p
if not found, lookup A as if it were free
```

The latter approach is equivalent only if A is a simple identifier. However, A might involve expression evaluation:

```
template <class T, int i> class other_template {
    struct S { int operator -(int); } j;
    };
template < class T, int j > class Q: other_template<T, i> {
    void f() {
        this->other_template<T,j-1>::m = 1;
        }
    };
```

Example 13

Here other_template, T, i, and the operator - are evaluated in the context consisting first of the class scope of *this, then as if free where written. In this particular case, we see that j in j-1 comes from other_template, not from the int parameter to class Q; and, - is overloaded for j.

operator T.

In core-1479, Scott Turner asked that we explore the relationship between Anthony Scian's "x.operator T" problem and the x.B:: solution. Here is an example that Scott presented in -1479:

```
typedef int *T;
struct B {
    typedef  char  *T;          //1
    operator int   *();
    operator char  *();         //2
    operator short *();
    operator long  *();
    };
struct D : B {
    typedef short  *T;
    operator char  *();
    };
void *foo (D &d) {
    typedef long *T;
    return d.B::operator T();   //3
                // which T?  which conversion function?
    }
```

The problem is that B::operator T is not simply a lookup of the name "operator T" in B. Names of the form "operator T" require evaluation before they turn into names that can then be looked up.

We propose the following:

Operator Rule. In an expression of the form y :: operator conversion-type_id x . operator conversion-type-id p -> operator conversion-type-id conversion-type-id is evaluated in a context consisting of - first, the class scope denoted by y (x, *p) - second, the scope containing the expression (i.e., as if conversion-type-id appeared free at the point of its writing).

Operator Rule is basically the same as Rule. The answer can now be made to Scott's example. In //3, T is evaluated in the scope of B, obtaining char * due to the typedef in //1. The result is that the operator char * from //2 is chosen.

Acknowledgements. My thanks to Scott Turner for careful commentary on a first draft of this paper.

```
struct B { };

struct A:B{
    struct B { static int x; };
    void f(A*p) {
        B::x = 0;
        this->B::x = 0;
        A::B::x = 0;
        p->B::x = 0;
    }
}
```

Is it simple?
Can it catch a non-base?
Can it catch a non-class member?