Doc No: X3J16/96-0046   WG21/N0864
Date: January 30, 1996
Project: Programming Language C++
Ref Doc:
Reply to: Josee Lajoie
(josee@vnet.ibm.com)

Access Issues and Proposed Resolutions

586 - When do access restrictions apply to default argument names?

Neal Gafter asked the following:

```
>   class C {
>           static int f() { return 0; }
>   public:
>           C( int = f() ) { }
>   };
>   C c; // error? C::f accessible?
>
>   class D {
>           static int i;
>   public:
>           D( int = i ) { }
>   };
>   D d; // error? D::i accessible?
>
>   Does access checking take place when the default argument name is
>   bound (at the point of the function declaration) or when the
>   default argument name is implicitly used on the call?
```

Proposal:

When the default argument name is bound.

Add to 8.3.6[dcl.fct.default], at the end of paragraph 5:

"Access checking on a name used in default arguments takes place when the name is bound. [Example:

```
class C {
    static int f() { return 0; }
public:
    C( int = f() ) { }
};
C c; // well-formed
```

-- end example]

585 - Is access checking performed on the qualified-id of a member declarator?

11[class.access] paragraph 6 says:

"It is necessary to name a class member to define it outside of the definition of its class. For this reason, no access checking is performed on the components of the qualified-id used to name the member in the declarator of such a definition. [Example:

```
class D {
    class E {
        static int m;
    };
};
int D::E::m = 1;   // Okay, no access error on private 'E'
```

-- end example]

Unfortunately, the paragraph above also makes the following code

well-formed:

```
    class D { D f(); };
    class C
    {
        typedef D T;
    };

    D C::T::f() {} // Legal? T is a private typedef of C.
```

Proposal:
=========
Change 11[class.access] paragraph 6 to say:
"It is necessary to name a class member to define it outside of the definition of its class. For this reason, the qualified-id used to name the member in the declarator of such a definition can refer to the names of classes enclosing the member's class definition, even if these classes are private or protected members of their enclosing class.
[Example: /* same as current example in paragraph 6 */]"

388 - Access and qualified ids

11.3 [class.access.dcl] paragraph 1 says:
"The base class member is given, in the derived class, the access in effect in the derived class declaration at the point of the access declaration."

Jerry Schwarz asks:

```
>   It isn't clear what this means for
>       class B { public: int i; };
>       class D : private B {
>       public:
>           using B::i;
>       };
>
>       main() {
>           D* p;
>           p->i; // clearly well-formed
>           p->B::i; // is this well-formed?
>       }
```

Proposal:
=========
This should be part of the semantics description for using declarations.

Move the text above to 7.3.3 Using Declarations, [namespace.udecl], at the end of paragraph 3 and add:
"..., even if the member name is qualified by the base class name.
[Example: /* add Jerry's example above */]"

515 - How can friend classes use private and protected names?

11.4 [class.friend] paragraph 2 says:
"Declaring a class to be a friend implies that private and protected names from the class granting friendship can be used in the class receiving it.
[...]

Erwin Unruh mentioned:

```
>  This is not very explicit.
>  Where can the private and protected names be used in the befriended
>  class?
>  In the base classes of the befriended class?
>  In the nested classes of the befriended class?
```

Proposal:
=========
The sentence above should be replaced with:
"Declaring a class to be a friend implies that the names of the private and protected members of the class granting friendship can be used in the definition of the class receiving friendship (excluding in the definition of nested classes of the class receiving friendship) or, if a static member or a member function of the class receiving friendship is defined outside of its class definition, in the definition for this member, after the member declarator."
and delete the last sentence of paragraph 2.

441 - How do access restrictions apply to base class names?

```
class C {
    class A { };
    class B : A { }; //1
};
```

Is the declaration on line //1 ill-formed because the nested class B cannot refer to the private type A declared in C?
Or is it well-formed because the name A can be used in the scope C?

Since names used in a class definition after the declarator for the class is seen (this includes the names used in a base-clause) are looked in the scope of the class being defined (3.4.1[basic.lookup.unequal], paragraph 6), it seems that access restrictions for the names used in a base-clause should also be checked as if the names were referenced from within the scope of the class being defined. This implies that line //1 above is ill-formed because a nested class cannot access the name of a private member of an enclosing class.

Proposal:
=========
Add to 10[class.derived] paragraph 2:
"In the definition of a class, access restrictions apply to names used in the base-clause as if these names were used in a member function of that class."

532 - Is a complete class definition allowed in a friend declaration?

Neal Gafter asks:

```
>   Is this allowed:
>
>       class A {
>           static int x;
>           friend class B {
>               int f() { return A::x; };
>           };
>       };
>
>   If so, what is the scope of the class name B?
```

Solution 1):

It is disallowed.
This makes the name look up rules for such classes rather simple to describe.

Solution 2):

It is allowed. In which case we have to provide a description for how

name look up proceeds in the definition of class B. Possible solution, copy what 11.4 paragraph 5 says for friend functions defined in the friend declaration.

"A class can be defined in a friend declaration of a class if and only if the friend class name is unqualified. A class defined in a friend declaration of a class is in the (lexical) scope of the class granting friendship."

Proposal:
=========
I prefer solution 1) for simplicity.
I can live with either.

625 - Can a friend function be declared "inline friend"?

Is the following allowed?

```
class C {
    inline friend void f();
};
void f() { }
```

What is the linkage of such a friend function?
Does "inline friend" mean the same thing as "extern inline"?
Or does the fact that the function is inline mean that the function receives internal linkage?

Solution 1):

A friend function explicitly declared inline in the friend declaration is an inline function with external linkage.

Solution 2):

Explicitly declaring a friend function "inline" is ill-formed. I don’t believe it makes sense to say that the inline specifier gives the friend function internal linkage because it doesn’t make sense that the declaration of an entity with internal linkage be allowed within a class with external linkage.

Proposal:
=========
I prefer solution 1).
I can live with either.