On Entering Names into Scopes.

Bjarne Stroustrup

Consider

```
class X * p;
friend class Y;
class Z;
```

Each introduce a class name into a scope. Which scope?
I think

```
class Z;
```

is the easy case. It introduces 'Z' into the scope in which it occurs
hiding any Z in enclosing scopes.

```
class X * p;
friend class Y;
```

differ in that they either match a name already entered in an enclosing scope
or they enter their name in some enclosing scope. Which enclosing scope?

A similar need to introduce a name into some enclosing scope occurs when a
function is declared friend:

```
friend void f();
```

First suggestion: The last three kinds of declarations should behave identically
as far as matching names and entering names into enclosing scopes go.
Anything else would be confusing.

Second suggestion: For such a class declaration we first search enclosing scopes
for a matching class name using the usual lookup rules and if we find one then that
is the name referred to. For such a function declaration we first search enclosing
scopes for a matching names using the usual lookup rules and if we find one then
that is the name referred to. For example:

```
class X { /* ... */ };

class C {

        class Y { /* ... */ };

        friend class X; // finds the X above

        class Y * p;    // finds the Y above
};
```

and:

```
class X { /* ... */ };
```

```
class C {

        typedef int X;

        int Y;

        friend class X; // finds ::X above

        class Y * p;    // finds ::Y below
};

class Y { /* ... */ };
```

This looks pretty sick and I'd have preferred to have 'class X' and 'class Y' match the local names and be errors, but if we remove the declarations involving 'X' and change 'class' to 'struct' we are left with a strictly conforming ISO C program that it could be an embarrassement to reject.

To contrast 'class Y;' enters a name in its scope and hides and 'Y's in enclosing scopes. For example:

```
class Y { };

class X {
        class Y;
        int i;
        Y* p;           // finds X::Y
        class Y { };
};
```

whereas:

```
class Y { };

class X {
        int i;
        Y* p;           // finds ::Y
        class Y { };
};
```

If we don't find a matching name for

```
        friend class X;
        class Y * p;
```

which enclosing scope should the new class name be entered into?

I see three plausible answers:

[I] The immediately enclosing scope
[N] The nearest enclosing non-class scope
[G] The global scope

[G] was the original choice for C++. [I] is the definition in the ARM and should therefore be retained unless there is a good reason not to. Whichever rule we chose we enter the name into the global scope if the

declaration occurs in the global scope.

The reason for the change from [G] to [I] was that [G] it seemed to encourage unnecessary name space pollution. Consider

```
class X {
        void f()
        {
                class Y * p;
        }

        class Y { /* ... */ };
}
```

Under rule [G], a 'Y' would be entered into the global scope and 'p' would point to an object of the global class Y. Under rule [I], only X::Y would be used and no name would be entered into the global name space. If we chose rule [G], we would have to write:

```
class X {
        class Y;

        void f()
        {
                class Y * p;  // finds X::Y
        }

        class Y { /* ... */ };
}
```

In this example rule [N] behaves like rule [G] and both seems wrong to me in that they require explicit action on the part of the programmer to maintain locality. Typically a programmer wouldn't notice problems with locality until much later. Rule [I] seems to be the right default.

However, what if we don't want the default. Given rule [G] we can use 'class y;' to ensure locality. Given rule [I] we need to be explicit to break locality. Either:

```
class Y;

class X {

        void f()
        {
                class Y * p; // finds ::Y
        }

        class Y { /* ... */ };
};
```

or

```
class X {

        void f()
        {
```

```
                class ::Y * p; // finds ::Y
        }

        class Y { /* ... */ };
};

class Y { /* ... */ };
```

Unfortunately(?), the use of :: in a declaration is not allowed so

```
class ::Y * p;
```

is not legal. As shown that is not much of a problem, but the need to put the 'class Y;' declaration outside the class makes it inelegant and potentially troublesome in the case of templates.

Friend functions produce a similar example:

```
class X {

        class Z {
                friend void g();
                // ...
        };
};
```

With rule [G] Z will declare a global function g() as its friend. Given rule [I], the friend declaration will be an error because it is not allowed to inject a member name into a class's name space except by an explicit member declaration within the class. Again rule [N] gives the same result as rule [G].

However, consider a slightly different example:

```
void g();

class X {

        class Z {
                friend void g();
                // ...
        };
        void g();
};
```

Here rule [I] will let g() find X::g() and rules [G] and [N] ::g(). Again I think I prefer rule [I] as the default because it maintains locality and again I see this as an argument for accepting :: in a declaration:

```
void g();

class X {

        class Z {
                friend void g();        //finding X::g
                friend void X::g();     // explicitly X::g
                friend void (::g)();    // explicitly ::g()
```

```
                        // ...
                };
                void g();
        };
```

Note that I wrote (::g) instead of plain ::g(). In this case, it makes no difference but had g() returned a class object the parentheses would have prevented an ambiguity:

```
        class T { /* ... */ };

        class X {
                friend T ::g(); // friend int T::g() or
                                // friend T (::g)()
                friend T (::g); // unambiguous
        };
```

This is another case where the implicit 'int' causes mischief.

Consider also this example

```
        class X { /* ... */ };

        class C {
                class X { /* ... */ };
                friend class X;
                // ...
        };
```

Clearly, it is C::X that is declared to be a friend. With a small change we have to choose:

```
        class X { /* ... */ };

        class C {
                friend class X;
                class X { /* ... */ };
                // ...
        };
```

The answer to this one depends on the resolution to the more general question of name lookup in class scope.

Type names introduced in argument declarations constitutes a particularly important and nasty case. For example:

```
        class X {
                void f(struct Y*);
        };
```

In my experience, people who writes this expect 'Y' to be global. Where Y has bee previously used as in

```
        class X {
                struct Y* p;
                void f(struct Y*);
        };
```

and

```
struct Y* p;
class X {
        void f(struct Y*);
};
```

there is no problem.

Under rules [G] and [N] there is no problem except for people who expected a more local binding to occur (in my opinion the less common case). Rule [I] makes

```
class X {
        void f(struct Y*);
};
```

equivalent to

```
class X {
        struct Y;
        void f(Y*);
};
```

This at least ensures that people who depend on 'Y' being global are caught by the compiler in all but the rare case where the programmer both relied on Y being global and also later declared a local Y.

Note that this would allow a class name to be entered into the name space of a class. This would be an error unless that class was actually defined:

```
class X {
        void f(struct Y*);
}; // error X::Y not defined
```

or

```
class X {
        void f(struct Y*);              // Y is X::Y
        struct Y { /* ... */ };
};
```

or

```
class X {
        void f(struct Y*);              // Y is X::Y
}; // no error yet

struct X::Y { /* ... */ };      // proposed by Tony Hansen
```