#X3J16/95-0035,#WG21/N0635

Relaxing the Rules for Namespace::Member

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This note is a response to suggestions to allow `N::m` to refer to an `M` that isn't declared in `N`, but is accessible in `N` because of a *using-directive*. This notes suggests what I consider the minimal relaxation of the current rules that allows the desired notational convenience without adding new syntax or removing existing benefits. As an added benefit, the specification of the standard library would be simplified. Finally, I briefly discuss an alternative proposal based on the `N::*` syntax.

1 The Problem

I have repeatedly been asked to make this work:

```
namespace A {
        int f();
}

using namespace A;

void g()
{
        ::f(); // call A::f
}
```

and this

```
namespace A {
        int f();
}

namespace B {
        using namespace A;
}

void g()
{
        B::f(); // call A::f
}
```

Under the current rules, this doesn't work because `B::f` means ‘‘look for an `f` declared in `B`’’ and `f` isn't declared in `B`.

Consider:

```
void f(char);

namespace A {
        int f(double);
}

using namespace A;

void g()
{
        f(1);    // error: ambiguous
        ::f(1);
        A::f(1); // clearly not ::f(int)
}
```

Under the current rules, `::f` unambiguously names `f(char)`. Having explicit qualification available for qualification is a valuable property that I don't want to lose.

Similarly:

```
namespace A {
        int f(char);
}

namespace B {
        using namespace A;
        void f(double);
}

void g()
{
        using namespace B;
        f(1);   // error: ambiguous
        B::f(1);
        A::f(1); // clearly not B::f(int)
}
```

under the current rules, `B::f` unambiguously names `B::f(double)`.

2 A Relaxation

However, it appears that we can have both! Consider this suggestion:

(1) As before, `N::f(){...}` defines an `f` explicitly declared in `N` only; *using-directive*s have no effect on definitions.

(2) If an `f` is declared in `N`, `N::f` refers to that `f`; *using-directive*s are ignored.

(3) If no `f` has been declared in `N`, `N::f` identifies an `f` found through a using directive in `N` exactly as if `f` had been used within `N`.

(4) If the lookup specified in (3) leads to an ambiguity `N::f` is an ambiguity error.

I will elaborate below, but this ought to convey the central idea.

One could argue that this interpretation is closer to the way `B::f` always worked for a base class `B`.

A benefit would be a simplification of the library headers because

```
namespace std {
        int printf(const char* ... );
        // ...
}

using namespace std;

int main()
{
        ::printf("Hello pedantic world\n");
}
```

would now work. It this relaxation is accepted, I would expect the standard `.h` headers to be changed to use *using-directive*s (as originally intended) rather than *using-declaration*s. This would save hundreds of lines of declarations.

Also, if someone takes

```
static void f(char);
void f(int);

void g()
{
        ::f('a');  // calls f(char)
}
```

and naively translates it to

```
namespace { void f(char); }
void f(int);

void g()
{
        ::f('a');  // current rules: class f(int)
                   // relaxed rules: calls f(char)
}
```

then there would be a change of meaning under the current rules, but not under my suggested new rules. Some people have worried about the change of meaning implied by the current rules.

People have responded to this proposal with remarks like ‘‘obvius,’’ ‘‘that was what I always meant,’’ and ‘‘I thought that was what it did.’’ I consider that an indicator that the relaxation will not lead to added teaching problems, but might reduce such problems.

3 Implementation

An obvious implementation appears to be to apply the existing lookup mechanism for a name used within `N` when the current resolution of `N::f` fails to find an `f`. Unfortunately, I haven't had a chance to try that technique, but it seems as straightforward as they come.

4 Details

Allowing `N::m` to refer to an `m` not explicitly declared in `N` raises some questions about ambiguity. The general answer to those is that ‘‘if `m` is not explicitly declared in `N`, `m` is looked up as if it had been used within `N`.’’ This resolves the problem by reducing it to a previously solved one.

Consider

```
namespace A { int x; }
namespace B { int x; }
namespace C { using namespace A; using namespace B; }

void f()
{
        C::x++; // error, ambiguous: A::x or B::x?
}
```

and

```
namespace D { using namespace A; int x; }
namespace E { using namespace D; }

void g()
{
        E::x++; // error, ambiguous: D::x or A::x?
}
```

The reason is that is no `x` declared in `E`, so we need to see if any `x` is visible from `E`. We find both `D::x` and `A::x` so we have an ambiguity error.

In accordance with the rule that a *using-directive* makes names available in the context in which they were declared rather than by introducing local aliases, there are no hiding effects based on the order in which *using-directive*s are encountered.

As usual, overloading can occur, and explicit qualification can be used for explicit resolution. For example:

```
namespace A { int f(int); }
namespace B { int f(char); }
namespace C { using namespace A; using namespace B; }

void g()
{
        C::f(1);  // A::f(int)
        using namespace C;
        f(1);     // A::f(int)
        B::f(1);  // B::f(char)
}

namespace D { using namespace A; int f(double); }
namespace E { using namespace D; }

void h()
{
        E::f(1);  // A::f(int)
        using namespace E;
        f(1);     // A::f(int)
        B::f(1);  // B::f(char)
        D::f(1);  // D::f(double)
}
```

5 Suggested WP Text

We really ought to have WP text ready before Austin, but I have run out of time. Any volunteers? Please email me before starting (in case we have several volunteers).

6 Using N::*

Several people (me included) have stumbled upon the idea of having an operation meaning ''insert *using-declaration*s for every member of namespace `N`'' and all seem to have invented the notation `using N::*` for that operation. Tom Penello wrote a proposal along these lines: #?????.

Despite being one of the first (dozen?) to think of this, I dislike the idea and repeated consideration has strengthened my dislike.

First of all, the notation is (objectively) wrong, and nobody has suggested a better one. In C++, `*` means multiply, pointer, or dereference, and `X::*` is a notation for ''pointer to member.'' The fact that `*` means ''match any pattern'' in popular regular expression languages seems a poor excuse to overload a C++ term yet again.

The *using-directive* was designed to avoid errors due to name clashes due to unused names or multiple uses. For example:

```
namespace N {
        int x;
}

using namespace N;
using namespace N;
```

doesn't cause a name clash because no name from `N` is injected into the global scope. Further,

```
namespace M {
        int x;
}

using namespace N;
using namespace M;
```

doesn't cause a clash because no `x` is used. However,

```
using N::x;
using M::x;
```

does cause a clash, as would

```
using N::*;
using M::*;
```

provided `using X::*` is defined in the intuitive way as a shorthand for a *using-declaration* for every member of `X`.

Even

```
using N::*;
using N::*;
```

would cause a clash.

In fact, this is an example of a deliberate philosophical difference between *using-declaration*s and *using-directive*s. A *using-declaration* declares a local alias that behaves exactly like other local declarations as far as overloading, name clashes, etc. goes, whereas a *using-directive* makes names accessible from the context in which they were declared.

In particular, consider:

```
namespace M {
        void f(bool);
}

namespace N {

        void f(int);

        void g()
        {
                using namespace M;
                f(1);     // calls N::f
        }
}
```

Nothing in namespace `M` can hijack the call `f(1)`. Replacing, the *using-directive* by `using M::*;` would give presedence to names from `M` and make hijacking an everyday occurrence.

7 Acknowledgements

Many people contributed. In particular, I borrowed some of the examples in the ‘‘details’’ section from one of John Skaller’s reflector messages.