Document Numbers: X3J16/95-0043
WG21/N0643
Date: January 31, 1995
Reply To: Bill Gibbons
bgibbons@taligent.com

Overload Resolution for "using" Member Functions

The Problem

Given this example given by Mike Ball in `c++std-ext-2674:`

```
struct B {
    int f(int);
};

struct D : B {
    using B::f;
    long f(long);
    void g();
};

void D::g() {
    f(1);         // ambigous ?
}
```

Is the call to “f” ambiguous? According to the overload resolution rules, the “this” parameter is resolved as if it were an explicit parameter, so overload resolution should be the same as in the following example:

```
struct B { };
struct D : B { };

void f(B*, int);
void f(D*, long);

void g(D *d) {
    f(d, 1);      // ambigous
}
```

The second example is certainly ambiguous. Yet we believe that it was the committee’s intent that the first example not be ambiguous.

The problem is that when overload resolution is applied to the implicit “this” parameter,`D::f` is preferred to `B::f` because of the conversion from “`D*`” to “`B*`”. This is an accidental interaction between “using” and overload resolution.

The Solution

There was general agreement on the extensions working group reflector that the first example should not be ambiguous. This can be done by amending the overload resolution rules so that the “`D*`” to “`B*`” conversion has no effect on overload resolution. This can be phrased either in terms of how

overload resolution works, or how a “using” declaration is treated during overload resolution. There seemed to be some preference for the second approach.

So we propose adding the following paragraph after 7.3.3/12, subject as always to editorial improvement:

> For purposes of overload resolution, using-declarations which name members of base classes shall be treated as members of the class containing the using-declaration. In particular, the implicit “this” parameter shall be treated as if it were a pointer to the derived class, not the base class.

Effect on Existing Programs

This should have no effect at all on existing programs, since without “using” declarations all members of an overloaded set must be members of the same class (or not members of any class).

Pointers to Members

Since a “using” declaration does not affect the type of the referenced declaration, in the following example:

```
struct T {
    int f(int);
    void g();
};

struct U : T {
    using T::f;
};

int  (T::*fp1)(int) = &U::f;  // OK, in this proposal
void (T::*fp2)()    = &U::g;  // OK, as specified in working paper
```

the type of `&U::f` is “`int (T::*)(int)`”, because `&U::f` is an alias for `&T::f`, not a new member function of `U`. Just as the type of is `&U::g` is “`void (T::*)()`”, because it’s inherited (as specified in the current working paper). So both initializations are valid, even though they appear to be ill-formed. But what about:

```
struct B {
    int f(int);
};

struct D : B {
    using B::f;
    long f(long);
};

int  (B::*fp2)(int)  = &D::f;  // OK
long (B::*fp1)(long) = &D::f;  // error
```

Because there are two instances of `&D::f` (a member declaration and a using declaration), this example requires selecting a function from an overloaded set (13.3). But the overloaded set has some functions which, if selected, cannot be used because the corresponding pointer to member cannot be converted to the type of the variable to be initialized.

Is this a problem? No. Each individual function in the set is considered independently. The first initialization is OK because the selected function is `B::f(int)`. The second is ill-formed either because:

- The selected function is `D::f(long)`, so the initializer has type "`long (D::*)(long)`" which cannot be implicitly converted to the initializer type (because it would require a pointer to member upcast).

or

- There is no matching function.

The choice of reasons depends on how the committee rules on the "exact match" criteria for selecting functions from an overloaded set; but either way, there is no problem with the example.

Overloaded Function Selection

The above discussion brings up a related point which may or may not be editorial. It's included here to give everyone time to see it before the Austin meeting.

As written, 13.3 has a surprising ramification:

```
struct B {
      void f(int);
      void f(long);
};

struct D : B { };

void (D::*pf)(long) = &B::f;
```

is ill-formed. The wording in 13.3 is:

> A use of a function's name without arguments selects, among all the functions of that name that are in scope, the (only) function that exactly matches the target.

The problem is that `B::f(long)` does not match `pf` because the class is wrong.

Similarly, this is a potential problem:

```
void f(int);
void (*p1)(const int) = &f;
void (*p2)(int) = (void (*)(const int)) &f;
```

It should be clear that top-level const has no effect in either place.

I suggest rewording 13.3 as:

> A use of a function's name without arguments selects, among all the functions of that name that are in scope, the (only) function that has exactly the same number of parameters and parameter types, ignoring top-level qualifiers.

Any errors due to mismatching classes (for pointers to members) or return types would be caught by the usual initialization rules.