Class Name Injection

Doc No: **X3J16/94-0057**
WG21/N0444
Date: **May 31, 1994**
Project: Programming Language C++
Reply-To: Neal M Gafter
neal@cs.rochester.edu

1 Introduction

(1) There are a number of special contexts in which, for the purpose of name lookup, “the name, if any, of each class is also considered a nested class member of that class.” This quoted phrase, describing what I call *name injection*, is repeated in the Working Paper in each of these many contexts. I assert that name injection should not be restricted to these special contexts, but should apply uniformly. This would be a simplification of the Working Paper, and would define language behavior in a number of currently grey and counterintuitive areas. This proposal would resolve the two remaining open core issues concerning name lookup.

2 What contexts?

(1) Here are the contexts in which the lookup for `T` behaves as if we had name injection:

```
x.T::a             // expression
p->T::a            // expression
x.operator T       // expression
p->operator T      // expression
x.operator T::U    // expression
p->operator T::U   // expression
A::operator T      // expression or declarator
A::operator T::U   // expression or declarator
```

(2) Here is a context in which the Working Paper rules are confusing:

```
template <class X> struct list { int a; };

struct U {
      typedef int T;               //1
};

typedef struct T : U {             //2
} A;

struct X : public list<A>, list<int> {} x;

A::T t1;           // uses //1

x.list<A::T>::a    // uses //2
```

(3) Normally, the type `A::T` would designate `//1`. But in this latter context, `A::T` designates `//2` even though there is only one `A`. How can this be? Well, “...for the purpose of this type lookup, the name, if any, of each class is treated as a nested class member of that class.” This causes the name T marked //2 to be considered a member of its class, hiding the T marked //1. The additional text “If the *nested-name-specifier* contains a *template-class-id*, its *template-argument*s are evaluated in the context in which the entire *postfix-expression* occurs.” doesn’t cancel the required use of name injection in this context.

(4) Here is another counterintuitive case:

```
class N {

      class B {
            typedef class B A;
            int x;      //2
      };

      class A : B {
            int x;      //1
      };

      class D : A {
            int f();
      };

};

int N::D::f() {
      this->A::x; // uses //1
      A::x;       // uses //2
}
```

(5) This example is similar, but illustrates that the current name lookup rules violate the very principles on which they were built. Specifically, they violate the principle that, for an *id-expression* γ, if γ and `this->`γ are both well-formed, then they should mean the same thing. This principle is one on which the current rules were based, but don't fulfil.

(6) I propose to replace all of these special case rules with the single rule of *uniform name injection*. Uniform name injection says that "the name of each class is treated as a nested class member of that class." Period. The rule applies uniformly in every applicable context. The main arguments against uniform injection seem to have been that

(7) • Injection introduces a conflict with the name of constructors in the class scope. The conflict, however, is there today without uniform injection because of the special contexts in which injection is applied. See core issues 14 and 290. These are, by the way, the only remaining open issues in name lookup.

(8) • Injection doesn't appear to be universally agreed upon by compiler implementations. An informal survey indicated that roughly half of the available implementations use injection. The debate appeared to be polarized between these two camps. The special case rules seem to have been developed as a compromise between those who prefer injection and those who don't.

(9) Let us examine the two sides of the name lookup and injection issue and how the compromise was reached. I conclude that the compromise is more fundamentally flawed than either original alternative. I then suggest three alternatives. The first two alternatives are the original sides of the debate, and the final alternative is an "improved" (but unfortunately more complex) compromise. I suggest we adopt the name injection alternative, but all three are documented here for the committee to consider.

3 "We prefer no name injection"

(1) The basis for this argument is a consistent and completely workable model for treating class names and lookup in the contexts listed in "What contexts?" on page 1. The model works as follows:

(2) a) Each class name is inserted in the scope in which it was declared, and is not "injected".

(3) b) Constructor names are regular named members with some special properties.

(4) c) In a construct such as `p->A::x`, `A` is looked up in the context of the full expression.

(5) d) Within the scope of the class, constructors are “skipped” in most contexts when performing a name lookup, so that the name of the class can be found in the enclosing scope. Declarators are one of the few (only?) contexts in which this skipping doesn’t take place.

(6) These rules are simple, consistent, and easy to understand. Item (d) would resolve the open name lookup issues. A number of compilers work this way. This would be a sensible approach to all of the situations from “What contexts?” on page 1. A disadvantage would be that people who currently use compilers with name injection might get different results.

4 “We prefer name injection”

(1) This is *uniform name injection*. The basis for this argument is a consistent and completely workable model for treating class names and lookup in the controversial contexts. The model works as follows:

(2) a) Each class name is inserted in the scope in which it was declared, and additionally into the scope of the class itself.

(3) b) Constructors don’t have names, but are simply a special declarator syntax.

(4) c) In a construct such as `p->A::x`, `A` is looked up in the context of the class that `p` points to.

(5) d) Constructors are never found during name lookup. However, a special declarator syntax involving the class name is recognized for constructors.

(6) Note that you can’t name a constructor, a constructor can never be called, and its address can’t be taken. The only thing you can do with a constructor is to define it and invoke it indirectly by using a cast.

(7) These rules are simple, consistent, and easy to understand. Item (d) would resolve the open name lookup issues. A number of compilers work this way. This would be a sensible approach to all of the controversial contexts.A disadvantage would be that people who currently use compilers without name injection might get different results.

5 A flawed compromise

(1) The current compromise attempts to apply both philosophies, making the program ill-formed if the two strategies differ. The compromise work as follows:

(2) a) Each class name is inserted in the scope in which it was declared, and is additionally treated as if inserted into the scope of the class itself for the “controversial” contexts (see point c below).

(3) b) The compromise was undecided as to whether constructors have names.

(4) c) In a construct such as `p->A::x`, `A` is looked up both in the context of the full expression and in the context of the class that `p` points to. These lookups (only) are performed with injection, and must yield one result.

(5) d) The compromise was undecided as to whether constructors are found during name lookup.

(6) There are good reasons for wanting to treat lookup in `p->A::x` in each of these two ways. Looking in the scope of `p`’s class allows a name to appear the same within and outside of the context of the class. Looking in the local scope allows using a local typedef for A to simplify naming. However, the compromise is flawed in the following fundamental ways:

(7) • On two of these issues (b and d), the compromise is undecided. As a result, we have two open core issues concerning name lookup. We can repair this flaw by taking the decisions from either original alternative.

(8) • The compromise is much more complex than either of the basic alternatives, and very difficult to understand.

(9) • The compromise doesn't meet its design criteria. Specifically, the two extended examples in the first section of this paper are interpreted differently by the two basic strategies, but the examples are not ill-formed. In addition, the compromise violates the criterion that when γ and `this->`γ are well-formed, they will mean the same thing. Finally, people who currently use compilers that apply either of the two basic strategies can get different results when using an ANSI/ISO compiler.

6 An improved compromise

(1) I recommend (and propose) a solution that includes uniform name injection (""We prefer name injection"" on page 3). It seems to be the most clean solution to the problems. The improved compromise is:

(2) a) Each class name is inserted in the scope in which it was declared, and additionally into the scope of the class itself.

(3) b) Constructors don't have names, but are simply a special declarator syntax.

(4) c) In a construct such as `p->A::x`, `A` is looked up both in the context of the full expression and in the context of the class that `p` points to. These lookups must yield one result.

(5) d) Because they don't have names, constructors are never found during name lookup. However, a special declarator syntax involving the class name is recognized for constructors.

(6) This compromise keeps the best features of the original: it looks up `A` in `p->A::x` in both contexts. When γ and `this->`γ are well-formed, they will mean the same thing. However, folks in the "We prefer no name injection" camp will still notice differences from current compiler behavior for well-formed programs. I don't believe this problem can be fixed without making the compromise still more complex and difficult to understand. It is a problem shared by all proposed solutions.

(7) My second choice would be the following improved compromise, with the same properties. I believe this is a different model with the same effect.

(8) a) Each class name is inserted in the scope in which it was declared, and additionally into the scope of the class itself.

(9) b) Constructor names are regular named members with some special properties. The "C compatibility hack" allows the class name and the constructor names to coexist in the same scope.

(10) c) In a construct such as `p->A::x`, `A` is looked up both in the context of the full expression and in the context of the class that `p` points to. These lookups must yield one result.

(11) d) Within the scope of the class, constructors are "skipped" in most contexts when performing a name lookup, so that the name of the class can be found. Declarators are one of the few (only?) contexts in which this skipping doesn't take place.

(12) My third choice would be the original name injection alternative (""We prefer name injection"" on page 3); my fourth choice would be the no name injection alternative (""We prefer no name injection"" on page 2). These alternatives are equally clean. Although I prefer name injection, I would rather see my "opponents" get their clean solution than inflicting the existing flawed and complex compromise (name injection "sometimes") upon the C++ community.