Document Numbers: X3J16/96-0060
WG21/N0878
Date: March 11, 1996
Reply To: Bill Gibbons
bill@gibbons.org

Eliminating Name Injection (at template instantiation)

Introduction

Name injection as a side effect of template instantiation has long been recognized as being undesirable in most contexts, but necessary in a few restricted cases (see 95-0177/N0777). There have been several proposals for eliminating name injection, but these have been rejected on the grounds that they break existing code without providing alternatives for all of the most important cases.

This proposal provides such alternatives, and so does not break these crucial cases.

The Proposal

The key is in recognizing that a related problem has already been solved, namely lookup of operator function names in namespaces which are not normally searched.

A change was recently made to the working paper to add the "Koenig lookup rule" to 13.3.1.2 [over.match.oper]. This change extended the name lookup rules for operator function names so that the namespaces of the classes of the operands, and namespaces of the base classes of those classes, are included in the search for the operator function declarations. This is almost exactly what we need for friend function lookup. For example, a simple addition to this rule sufficient to eliminate name injection would be:

> In the lookup of a function name used in a function call (with either the function call syntax or operator syntax), the lookup shall include friend functions of the classes of the argument types, and friend functions of the base classes of those classes.

For example, prior to the Koenig lookup rule:

```
namespace A {
    class B {
        friend void operator - (B, B);
    };
    void operator + (B, B);
}
void f(A::B x, A::B y) {
    x + y;    // fails; no "operator+" found
    x - y;    // fails; no "operator-" found
}
```

With the current working paper (including the Koenig lookup rule):

```
namespace A { ... as above ...}
void f(A::B x, A::B y) {
    x + y;    // OK; "operator+" found via Koenig lookup rule
    x - y;    // fails; no "operator-" found
}
```

And with the proposed change:

```
namespace A { ... as above ...}
void f(A::B x, A::B y) {
    x + y;    // OK; "operator+" found via Koenig lookup rule
    x - y;    // OK; "operator-" found via friend lookup rule
}
```

The Two Main Friend Lookup Problems

This solves both of the serious problems which seemed to require name injection:

- Overloaded operators where the first operand requires conversion:

  ```
  template<class T> class complex {
      complex(T,T);
      complex(T);
      friend complex operator+ (complex, complex);
  };
  complex<double> a;
  complex<double> b = 2 + a;
  ```

  The lookup of "`operator+`" finds the friend because the operand "`a`" has class type "`complex<double>`", and so its friends are considered. So injection is no longer necessary to handle this case.

- The Barton and Nackman trick - in simplified form:

  ```
  template<class T> struct Equiv {
      friend bool operator==(const T &x, const T &y) {
          return x.equal(y);
      }
  };
  template<class T> struct Derived : Base, Equiv<Derived> {
      // ... usual stuff
      bool equal(const Derived &) const;
  };
  Derived<int> x, y;
  bool z = x == y;
  ```

  The lookup of "`operator==`" finds the friend because the operands "`x`" and "`y`" have class type `Derived<int>`, and so the friends of the base classes of `Derived<int>` are considered. Since base class `Equiv<Derived<int> >` declares the friend which needs to be found, the modified lookup rule finds that friend. So injection is no longer necessary here either.

Both of these examples involve operator functions, not ordinary functions. However, it seems useful to apply the rule to ordinary functions as well.

(The point has been made that there may be no functions with the appropriate name visible when the function call is parsed, so a compiler must decide that the name is going to be used as a function name before actually finding the name. But this is exactly the same case as for function name lookup in templates, when the call must be parsed prior to the phase-two name lookup. That case works because the grammar does not require lookup of nontype names prior to parsing, and this case works for the same reason.)

Variations

There are several possible variations of the friend name lookup rules, for example:

Type of lookup	current WP	option # 1	option # 2	option # 3
Operator functions & namespaces	yes	yes	yes	yes
Operator functions & friends	no	yes	yes	yes
Non-operator functions & namespaces	no	no	no	yes
Non-operator functions & friend	no	no	yes	yes

When considering the choice, we should account for this anomaly in the current working paper:

```
namespace B {
     struct A { };
     void operator+ (A,A);
 }
 void f(B::A x, B::A y) {
     x + y;           // works today because of 13.3.1.2
     operator+(x,y);  // does not work today
 }
```

The advantage of choice #3 above is that it makes the two expressions in the above example work the same way (unlike the current WP rules).

Impact on Existing Code

The affect on existing code is a change in the set of function declarations visible after the point of instantiation of a class template.

- No additional operator function declarations are visible (when looked up because the operator itself is used), because all operator functions which can be found by the friend lookup rule are currently found by injection or the Koenig lookup rule.

- Some additional non-operator function declarations are made visible by options 2 and 3, but this fixes an inconsistency in the Koenig lookup rule. Option 1 would leave these declarations invisible.

- Some friend functions which are visible under the current rules would become invisible under the friend lookup rules. For options 2 and 3, this only happens for functions which do not appear to operate directly on objects of the class type which declares the friends. For option 1 it also happens for some additional non-operator functions.

Non-Template Classes

Template and non-template classes currently both inject friend declarations; for template classes it is done at instantiation, and for non-template classes it is done at class declaration. If the proposed lookup rule applies only to template classes, there would still be a difference between template and non-template classes. This is neither an advantage nor a disadvantage of the proposed change.

However, it would be possible to apply the proposed rule to non-template classes too. That would make template and non-template classes very consistent with respect to friend lookup. This would be a significant advantage, although it has the potential for breaking some existing code.

It is also very simple to correct any programs which would break by replacing non-template injection with the friend lookup rule - simply add the missing function declarations just before the class declaration.

Conclusion

We have already agreed that we should remove name injection at template instantiation if possible. Here is a solution which provides the most important functionality of such injection by a simple generalization of an existing name lookup rule. We should take this opportunity to remove injection.

There are several variations on the proposed friend lookup rule. The most complete variation is “option 3” applied to both template and non-template classes. That would eliminate both forms of injection and make the Koenig lookup rule more consistent as well.

But even “option 1” applied only to template classes would be an improvement over the status quo.