Document Numbers: X3J16/95-0177
WG21/N0777
Date: September 26, 1995
Reply To: Bill Gibbons
bgibbons@taligent.com

# An Alternative to Name Injection from Templates

## Introduction

The current working paper specifies that when a class template is instantiated, the names of friend classes and functions are "injected" into an enclosing scope. This is intended to solve certain problems with function overloading, but has also been used for purposes not originally envisioned.

There are serious drawbacks to name injection which make it worthwhile to explore alternatives. One such alternative is a trivial change to the function template matching rules, as described below. The proposed solution solves the original problems which name injection was intended to address, but does not allow the additional tricks which have since been invented.

The incentive for removing injection came from discussions at the Monterey meeting where it was discovered that injection causes much worse problems than previously thought, and it would require considerable work just to make the concept well-defined. The discussion of the solution presented here was conducted on the core reflector beginning with message core-6034.

## The Original Problem

Consider a simple complex class template:

```
template<class T> class complex {
      complex(T,T);
      complex(T);
      complex operator + (complex, complex);
};
```

This is sufficient for handling certain kinds of operands, but not others:

```
complex<double> a, b;
complex<double> c = a + b;    // OK, uses complex::operator+
complex<double> d = a + 1.0;  // OK, operator+ and conversion
complex<double> e = 1.0 + b;  // ill-formed
```

With a non-template class, this problem can be solved by adding a global operator function:

```
DoubleComplex operator + (double, DoubleComplex);
```

That solution partially works for templates, as in:

```
template<class T> complex<T> operator + (T, complex<T>);
```

until you try to use types which aren't an exact match:

```
complex<double> f = 2 + b; // ill-formed: conflicting type deduction
```

One solution is to use the template instantiation itself to force an appropriate template to be created. Since it happens that this kind of global operator function must often be a friend of the class anyway, the friendship can be used to cause the name injection:

```
template<class T> class complex {
      complex(T,T);
      complex(T);
      complex operator + (complex, complex);
      friend complex operator + (T, complex);
};
```

Looking at the failing example again, we see that the instantiation of `complex<double>` causes the function

```
complex<double> operator + (double, complex<double>);
```

to be injected into global scope. This function is a match for the operands in the failing example, so the problem is solved.

Note that the `friend` keyword is used to trigger name injection only because it happens to be convenient: it already enables a global function declaration to appear in class scope, and it is an existing keyword.. A new storage class keyword, e.g. `inject`, would have been cleaner except for the problems caused by adding new keywords.

### Barton and Nackman

New features always seem to get used in surprisingly inventive ways, and this was certainly true for name injection. Barton and Nackman suggest using a class template to describe a set of global functions which are injected when that template is instantiated. They suggest this approach for global operators as an alternative to defining function templates which are too general and thus have unintended effects on the rest of the program.

While this technique is clever, it was not the reason why injection was originally proposed and it is not sufficient reason by itself to keep injection in the language.

### Why Injection Is a Bad Solution

As currently defined, injected names are "declared as if the specialization had been explicitly declared at its point of instantiation". This is both impossible and insufficient; it implies that in the following code:

```
void f() {
      2 + complex<double>(3,4);
}
```

there is some kind of equivalent program which looks something like:

```
void f() {
      2 +
            // interrupt the expression
            class complex<double> {
                  complex<double>(double, double);
                  complex<double>(double);
                  complex<double> operator +
                        (complex<double>, complex<double>);
                  friend complex<double> operator +
                        (double, complex<double>);

            };
            // note that the friend has been injected
            // resume the expression
      complex<double>(3,4);
};
```

Even if this concept were well-defined, it still doesn't solve the original problem: a compiler may decide to look up all global declarations of "`operator+`" at the point where the "+" operator was written, before the injection was done - and so the expression is still ill-formed because the injection was done too late to affect the operator.

At the very least, injection won't work unless the exact sequence of name lookups and injections is specified. And a strict left-to-right sequence does not work. This is a pretty severe implementation restriction; it is even worse when the two-step lookup rules for template bodies are considered.

Here's another ordering nightmare:

```
long f(int);

template<class T> class A
      friend char f(char);
      int g();
};

void g() {
      A<sizeof(f('x'))>(0).g() + A<sizeof(f('x'))>(0).g();
}
```

The first instantiation of `A` cannot occur until type of `f('x')` is known, so the lookup of `f` must occur first. That lookup finds "`int f(int)`", so the first instantiation is `A<sizeof(int)>`. The second instantiation cannot occur until the type of the second `f('x')` is known, but that lookup finds the injected "`char f(char)`" so the second instantiation is `A<sizeof(char)>`. So we have *two instantiations in the same expression with identical tokens but different types*. That is absurd.

This kind of inconsistency arises from doing injection in the middle of expressions. On the other hand, consider ordinary declarators. If declarator names were introduced immediately, it would lead to problematic cases such as:

```
int x[sizeof(x)];
```

The solution for declarators was to define the point of declaration to be after the complete declarator (but before the initializer). Would a similar solution work for name name injection?

Suppose injection were deferred until the end of the nearest enclosing complete expression.

Then the example would still have to be rewritten:

```
void f() {
      complex<double> dummy; // force injection now
      2 + complex<double>(3,4);
}
```

This might be acceptable when the instantiation is visible in the code, but it isn't always visible:

```
template<class T> dereferenceIncrementAndPrint(T *t) {
      print(1 + *t);
}
void f(complex<double> p) {
      dereferenceIncrementAndPrint(p);
}
```

There is no way to know when writing the template function `dereferenceIncrementAndPrint` that it might be used with a class which does injection and so must be handled carefully. So the template function would have to be written:

```
template<class T> dereferenceIncrementAndPrint(T *t) {
      // If "T" happens to be a template class which hasn't yet
      // been instantiated, force an instantiation of "T" now so
      // that any injection will be done before it may be needed.
      int dummy = sizeof(T);
      print(1 + *t);
}
```

I don't think it's necessary to ask if we want to require that people write code like this.

The bottom line is this:

- Injection doesn't really solve the original problem unless it is done immediately and affects other lookup within the expression - even some which are lexically before the point of instantiation.
- Immediate injection requires a very detailed and potentially confusing set of rules about sequences of lookups and instantiations.
- Even with such rules, immediate injection can cause very strange results.
- Therefore, injection caused by instantiation should be avoided if at all possible.

There are other points, involving the stability and maintainability of code, invariance of semantics when the order of definitions is changed, etc. But I think the above arguments are sufficient.

## An Alternative

There is another solution to the original problem: refine the template function parameter matching rules so that it is possible to write global template functions which work for classes like `complex`.

The crux of the problem is that some function parameters are actually needed for deducing the template type arguments, and some should just follow along but not participate in type deduction. In the above example, we need to write:

```
template<class T> complex<T> operator + (T, complex<T>);
```

in such a way that only the second argument is used for type deduction. Then when we write:

```
complex<double> b;
complex<double> f = 2 + b;
```

the type `T` is deduced as `double` based on the second argument. There is no ambiguity.

How can this "non-deducing" parameter attribute be specified? One way is with a keyword; without adding new keywords, the simplest choice is `explicit`:

```
template<class T> complex<T> operator + (explicit T, complex<T>);
```

This says that the type of the first parameter isn't determined implicitly by type deduction, but by explicitly specifying the parameter type.

Another possibility is to take advantage of the default argument mechanism, by saying that template parameters which make use of default type arguments are not deduced:

```
template<class T, class U=T> complex<T> operator + (U, complex<T>);
```

Since the first function parameter makes use of default type argument `U`, it does not participate in type deduction.

At first glance default arguments seem like an odd solution to this problem, but the current behavior of default arguments happens to be almost exactly what is needed.

Consider how explicit template function type arguments are used: they disable type deduction for the explicitly specified type parameters. Under the existing working paper, if you write:

```
template<class T, class U> complex<T> add(U, complex<T>);
complex<double> b;
complex<double> f = add<double, double> (2 + b);
```

you get exactly the desired effect, *according to the existing template rules*.

Now add a default type argument:

```
template<class T, class U = T> complex<T> add(U, complex<T>);
complex<double> b;
complex<double> f = add<double> (2 + b);
```

This also does what we need, *by a reasonable interpretation of the existing rules*.

Now let the first type argument be deduced:

```
template<class T, class U = T> complex<T> add(U, complex<T>);
complex<double> b;
complex<double> f = add<double> (2 + b);  // (1)
complex<double> f = add         (2 + b);  // (2)
```

Shouldn't lines (1) and (2) behave exactly the same way, as long as the deduced first type parameter in (2) is the same as the explicit first type parameter in (1)?

If so, then this use of default type arguments is essentially supported by the current working paper. It only needs some (not quite editorial) clarification.

## Proposal

We propose removing name injection from templates, and changing function template type deduction to improve matching of global function templates.

Working paper changes:

(1) Replace 14.2.4, which currently reads:

> Names that are not template members can be declared within a template class or function. When a template is specialized, the names declared in it are declared as if the specialization had been explicitly declared at its point of instantiation. If a template is first specialized as the result of use within a block or class, names declared within the template shall be used only after the template use that caused the specialization. [Example:... --end example]

with:

> Friend declarations and elaborated type specifiers within templates do not introduce any new declarations outside the template.
>
> Any friend declarations within a template must match an existing class or function declaration. When a friend declaration does not depend on any *template-argument*, the matching class or function must be declared prior to the template definition. When a friend declaration does depends on a *template-argument* , the matching class or function must be declared prior to the point of instantiation.
>
> If an elaborated type specifier does not match an existing declaration and the scope into which the name would be injected (the nearest enclosing non-class non-prototype scope) is outside the template, the program is ill-formed.

(2) Replace this sentence from 14.10.2/2:

> Type deduction is done for each parameter of a function template that contains a reference to a template parameter that is not explicitly specified.

with:

> Type deduction is done for each parameter of a function template that contains a reference to a template parameter that is neither explicitly specified nor has a default argument value.