X3J16/96-0054
WG21/N0872
March 12th, 1996
John H. Spicer, Edison Design Group

Name Leakage and the template compilation model

1. Introduction

The purpose of this message is to respond to the statements made about the "name leakage" problem attributed to the "inclusion" compilation model. In this message I will illustrate why the "name leakage" issue has been a fact of life in C++ far longer than templates have been around, and why the separation model actually suffers from far more serious "name leakage" problems. A "feature" of the separation model actually permits names to leak between translation units.

2. Why the “name leakage” problem is a red herring

In the inclusion model, the "name leakage" that occurs is nothing more than the same issue that has affected class definitions and inline function definitions in header files as long as C++ has existed. In other words, code in header files that is included in multiple files can mean different things if the declarations that precede that code vary from one file to another.

The following is basically the name leakage example from Bjarne's reflector posting, that I've modified to include the definition and use of my_class. I've also modified it so that use2.c includes sum.h, and sum.h includes sum.c as would typically be done in the inclusion model.

```
// sum.h:

void notify(int);


template<class T> T sum(vector<T>& v)

struct my_class {
        void my_member()
        {
                notify('a');
        }
};
#include <sum.c>

// sum.c:

template<class T> T sum(vector<T>& v)
{
        T r = 0;
        for (int i = 0; i<v.size(); i++) r = r + v[i];
        notify('a');
        return r;
}

// use2.c:
```

```
void notify(char);

#include<complex.h>
#include<sum.h>

void f(vector<complex>& vc)
{
        complex c = sum(vc);
        // ...
}

void n()
{
        my_class        mc;
        mc.my_member();
}

void g() {
        // ...
        notify(1);
}
```

Bjarne's concern is that the behavior of the sum template is changed by the declaration of notify(char). While I agree that this is unfortunate, it is also a fact of life in C++ as illustrated by the fact that a normal (e.g., nontemplate) inline member function (my_class::my_member) suffers from exactly the same problem. Bjarne is further concerned that the behavior of g() is changed by the declaration of notify(int) that is included. Once again, this is unfortunate, but fixing it would take far more than a different template compilation model.

The name leakage problem affects the following kinds of declarations in each of the two template compilation models:

	separation	inclusion
class declarations	yes	yes
inline functions	yes	yes
class templates	yes	yes
inline function templates	yes	yes
noninline function templates		
nondependent names	no	yes
dependent names	yes	yes

So, the separation model eliminates name leakage in one of the contexts in which it occurs, but does nothing about all of the other contexts. I'm not aware of any proposal to try to get rid of name leakage in the other contexts, and have seen no explanation why it acceptable in all of those other contexts (including dependent names in function templates) but is such an awful problem for nondependent names.

3. The separation model has a far more severe problem

The separation model actually suffers from a form of name leakage far more dangerous and difficult to deal with than the one that advocates of the separation model attribute to the inclusion model.

The separation model actually allows names to "leak" from one translation unit to a completely different translation unit.

3.1 Introduction to "merged contexts" or "name lookup across translation units"

This is a consequence of the rules for handling instantiations caused by other instantiations. When an instantiation takes place in the separation model, it takes place in a "merged context" that includes names from both the translation unit where the template is defined (let's call it D1) and the translation unit where the template was referenced (let's call it R1). Let's call this merged context M1. If the template then causes the instantiation of another template defined in a third translation unit (let's call it D2), the merged context (let's call it M2) includes the names from D2 plus the names from the context containing the template reference, which in this case is M1 (the merged context containing names from R1 and D1). This process continues as long as you have templates that themselves call other templates. If this continues for 10 levels, you end up dealing with a merged context that includes 10 different translation units.

Some people prefer to describe this as a name lookup process that takes place across several translation units, instead of calling it context merging. It really doesn't make any difference, because the results are the same no matter what name you choose to give it.

3.2 Proliferation of instantiation contexts

An interesting property of the separation model is that it creates a proliferation of the contexts in which the instantiation of a given function can take place. The number of possible instantiation contexts depends on the various paths through a call graph of the program that can be taken to reach a given template, except that only call graph nodes that represent functions generated from template are significant.

Consider this example:

```
f1.c
----
template <class T> void f1(T);
template <class T> void f2(T);
template <class T> void f3(T);

template <class T> void f1(T t)
{
        f2(t);
        f3(t);
        g(t);
}

f2.c
----
template <class T> void f1(T);
template <class T> void f2(T);
template <class T> void f3(T);

template <class T> void f2(T t)
```

```
{
        f1(t);
        f3(t);
}

f3.c
----
template <class T> void f1(T);
template <class T> void f2(T);
template <class T> void f3(T);

template <class T> void f3(T t)
{
        f1(t);
        f2(t);
}

int main()
{
        f1('a');
        f2('a');
        f3('a');
}
```

In this example, when using the separation model, there are either 6 or 8 different possible contexts in which f1(char) can be instantiated.

They are:

- f2.c
- f3.c
- f1.c + f2.c
- f1.c + f3.c
- f1.c + f2.c + f3.c
- f1.c + f3.c + f2.c
- f2.c + f3.c
- f3.c + f2.c

The reason I say 6 or 8 is that it is possible that the context merging process is commutative (e.g., that f1+f2 is equivalent to f2+f1). Because the context merging process is unspecified, it is impossible to answer this question.

The proliferation of instantiation contexts makes the already difficult process of trying to do ODR checking even more expensive when the separation model is used.

Under the inclusion model, each template has, at most, one instantiation point for each translation unit in which it is referenced. In the separation model, on the other hand, the number of instantiation contexts is dependent on the call graph of template based functions. Because adding nodes has the effect of multiplying the number of possible paths through the graph, it is very easy to have very large numbers of possible instantiation contexts. For example, by adding one more template based call site for both f2 and f3, the number of possible instantiation sites doubles from 8 to 16. In real applications that make heavy use of templatized data types and libraries like STL, the number of possible instantiation sites could easily reach the thousands

3.3 Why this introduces leakage between translation units

Consider the following source file

t1.c

```
void h(char);

template <class T> void f(T t)
{
        h(t);
}

template <class T> void g(T t)
{
        f(t);
}

void i()
{
        g(1);
}
```

This file requires the instantiation of g(int) and f(int). So, which function h will be called when f(int) is instantiated? The answer that most people would expect is h(char).

But suppose you link this file with another file t2.c:

t2.c

```
template <class T> void g(T);

void h(int);

int main()
{
        g(1);
}
```

And suppose that the implementation chooses to instantiate g(int) based on the call found in main() instead of the call found in i(). The instantiation context for f(int) will now be the context of t2+t1 instead of just being the context of t1. Now, f(int) calls h(int) instead of h(char) despite the fact that h(int) is not declared in t1.c and the only call of f(int) in the entire program is from within file t1.c!

At least with the inclusion model, you can see the complete set of names by looking at the preprocessed source of a given translation unit.

4. Conclusion

The inclusion model may not be perfect, but at least it is proven and doesn't introduce new classes of name leakage problems that never existed before.

The more we look into the details of the separation model, the more serious problems are found. What we have found so far s probably only the tip of the iceberg. If the separation model follows the pattern established by every other significant language feature that has been added, most of the problems won't be found until the model has actually been implemented.