Doc. no. WG21/N1448
J16/03-0031
Date: 05 April 2003
Reply-To: Mat Marcus
Adobe Systems
801 North 34th Street
Seattle, WA 98103-8882
Fax: 206 675 6825
Email: mmarcus@emarcus.org
and
Gabriel Dos Reis
INRIA Sophia Antipolis
2004 route des Lucioles — BP 79
06902 Sophia-Antipolis — France
Fax: 334 92 38 79 78
Email: gdr@acm.org

Controling Implicit Template Instantiation

This paper proposes an extension to the explicit template instantiation mechanism that would make it possible to control where a template is instantiated, even when implicit instantiation would have otherwise been possible under the inclusion model.

1 Need for Suppressing Template Instantiation

In industrial settings it is often desirable to be able to control the location of template instantiations. Consider for example a system that consists of an executable dynamically linked with a large number of shared libraries. If the compiler is allowed to implicitly instantiate commonly used template specializations then template code bloat — or even duplicate symbol definitions at link time — can ensue. Even in systems without shared libraries such control can be desirable in order to reduce build times. Consider this simplified example:

Example: Template code bloat. Let's say that we have a system with one executable named `Consumer` and one shared library `Supplier`. In the source for `Supplier` we have a file supplier.cpp that instantiates `MyVector<int>` and provides it for use by clients that dynamically link to `Supplier`

--- MyVector.h ---

```
// use inclusion model
template <class T>
```

```
   class MyVector {
     // some stuff goes here
   };
```

--- Supplier.cpp---

```
#include "MyVector.h"
// MyVector<int> is to be made available
// to clients of the shared library
template class MyVector<int>;
```

Now, consider the executable, `Consumer`: it dynamically links to `Supplier`. In the source for `Consumer` we have various source files that use `MyVector<int>`, one of which is consumer.cpp:

--- Consumer.cpp ---

```
#include "MyVector.h"
void foo(MyVector<int>& v)
{
 // use v here, causing some implicit
 // instantiation of MyVector<int>.
}
```

But we don't want implicit instantiation to occur here. Instead we want to share the code provided by the `Supplier`.

What options do we have today? We could separate the header into two: one piece for declarations only, and one for definitions. But that hardly leads to a maintainable solution in the case of header files that are provided by the standard library or other third party libraries. And even if that were feasible, there could be other compiler generated artifacts such as vtables whose instantiation cannot portably be controlled using this idiom.

2 Absence of Syntax to Express Semantics

Usually, a forward declaration helps to break circular name use dependencies. For ordinary function, it takes the form

```
int foo(int);
```

So, one might think that just putting the `template` keyword in front of a forward-declaration might express a "forward declaration of a function template specialization", but it does not; the reason is that the declaration

```
template int bar(int);
```

is the syntax to request an *explicit instantiation*, not that of a mere declaration.

So what to try next? Putting the keyword `extern` before the explicit instantiation syntax in order to suppress the actual definition is not currently allowed by 7.1.1/1

> [...] A *storage-class-specifier* shall not be specified in an explicit specialization (14.7.3) or an explicit instantiation (14.7.2) directive.

Finally, one might think that sticking a pair of angle-branckets after the `template` keyword might express the original intent, but it does not; the reason is that now

```
template<> int bar(int);
```

is declararing an *explicit specialization*, i.e. something different.

What is needed is a syntax to say "*do not implicitly instantiate this specialization here*". A workable solution would be to instantiate the template *once* in a common shared library shared by the others. This proposal satisfies the following extension desirability criteria

- improve support for library building – Yes
- improve support for generic programming – Somewhat
- remove embarrassments — Maybe — doesn't it look like `extern template` should just work?

3 Proposed Resolution

3.1 Basic Cases

We propose to address the problems mentioned in the preceding section by allowing explicit template instantiations to be declared as `extern`. For example, we might add an explicit extern instantiation of `MyVector<int>` to consumer.cpp in the example above to obtain.

```
--- Consumer.cpp ---
#include  "MyVector.h"

// Suppresses implicit instantiation below --
// Will be explicitly instantiated later.
extern template class MyVector<int>;

void foo(MyVector<int>& v)
{
   // use the vector in here
}
```

In this case the "`extern template`" declaration directs the compiler not to instantiate `MyVector<int>`, in consumer.cpp.

3.2 Advanced Cases

In the interest of offering users finer control of template instantiation under the inclusion model, there is another issue that is worth mentioning. Compilers vary in their aggressiveness when instantiating associated types to a given template instantiation. Some take a shallow approach while others try to recursively instantiate all related types. The extern template extension does finally give users a means to prevent instantiation of these associated types, but it can require a good deal of manual labor. Perhaps it would be worth spending some additional time to try to reduce the user's burden in this case.

In the case of an extern explicit class template instantiation, it is desirable that generation of *all* other class related artifacts be suppressed, e.g. vtables.

4 Interactions and Implementability

4.1 Interactions

The "extern template" construct, as proposed by the current paper, is meant to be applicable only to instantiations of non-exported templates. We believe that, except for the greedy instantiation issue, controling implicit template instantiation is less of an issue for exported templates.

This extension provides a syntax to express semantics that already exist for non-template functions, but is missing for specializations of non-exported templates:

- Remove the last sentence (quoted in §2) from 7.1.1/1.
- Change 7.1.1/5 to:

  The `extern` specifier can be applied only to the names of objects, functions and explicit template instantiations. The `extern` specifier cannot be used in the declaration of class members or function parameters. For the linkage of a name declared with an `extern` specifier, see 3.5. When used to declare an explicit instantiation, the declaration shall appear at namespace scope (3.3.5). The `extern` specifier used to declare an explicit instantiation of a class template only suppresses explicit instantiations of definitions of member functions and static data members not previously specialized in the translation unit containing the declaration.

The `extern` storage class specifier is usually viewed as the "opposite" of `static`; this proposal does not make any provision for the semantics of applying the storage class specifier `static` in lieu of `extern` as discussed above.

4.2 Implementability

This feature has been implemented in compiler front ends offered by Edison Design Group, GCC, IBM, Metrowerks, and Microsoft.

5 References

This proposal also addresses an issue raised by Steve Clamage on the Core Working Group reflector — see message `c++std-core-9243`.