Document number: J16/03-0103 = WG21 N1520
Date: 19 September, 2003
Reply to: William M. Miller
The MathWorks, Inc.
wmm@world.std.com

Extended `friend` Declarations

I. The Problem

According to the current Standard, 11.4¶2,

> An *elaborated-type-specifier* shall be used in a friend declaration for a class. *[Footnote:* The *class-key* of the *elaborated-type-specifier* is required.*]*

Unfortunately, the requirements for *elaborated-type-specifier*s, as given in 7.1.5.3¶2, place serious limits on the use of `friend` declarations:

> If the *identifier* resolves to a *typedef-name* or a template *type-parameter*, the *elaborated-type-specifier* is ill-formed. [*Note:* this implies that, within a class template with a template *type-parameter* T, the declaration
>
> ```
> friend class T;
> ```
>
> is ill-formed.]

Complaints about this restriction surface periodically in public discussion forums. One example of a use for this feature was posted by Hyman Rosen:

```
template <typename T>
class no_children {
      no_children() { } // private
      friend class T;   // ill-formed but desired
};

class bachelor: virtual no_children<bachelor> {
      /* … */
};
```

The `no_children` class template, if permitted, would provide an easy way of specifying that a particular class is “final” and cannot be used as a base class.

Gabriel Dos Reis (in message c++std-ext-6132) mentioned that various programming idioms such as the “visitor pattern” would benefit from this ability, and (in message c++std-ext-6141) posted two additional illustrations of its utility. The first involves an attempt to grant `friend` access to an allocator class:

```
template <class Allocator>
class X {
      friend class Allocator::template rebind<X>::other;
      // …
};
```

Here, the issue is not a template parameter but a typedef, which also cannot be used in an *elaborated-type-specifier*.

Dos Reis' second example illustrates yet another problem that can be encountered with `friend` declarations in template classes:

```
template <class T>
class Y {
      friend class T::U;
};
```

This template can be instantiated with any class that has a nested class named `U`, but the instantiation will fail if `T::U` is a union instead of a class: the *class-key* in the *elaborated-type-specifier* will disagree with that of the type to which it refers (7.1.5.3¶3).

II. History

In message c++std-ext-6133, John Spicer reviewed the Committee's deliberations that led to the current rule for *elaborated-type-specifier*s:

> Whether or not "`friend class T`" should be permitted was discussed in San Jose in November of 1993, where it was agreed that this should be permitted. Then in Waterloo, in July of 1994, we reopened the issue to address cases like this:
>
> ```
> template <class T> void f(struct T t){}
> template <class T> void f(union T t){}
> template <class T> void f(enum T t){}
>
> union U {};
> struct S {};
> class C {};
> enum E {};
>
> int main() {
>       U u;
>       S s;
>       C c;
>       E e;
>
>       f(u);
>       f(s);
>       f(c);
>       f(e);
> }
> ```

The issue being whether or not you can choose a function template based on the kind of *elaborated-type-specifier* that is present (something actually supported by the EDG front end back in those days). The committee decided against this, and as a result, a rule was added prohibiting a template parameter from being used in an *elaborated-type-specifier* in a function template declaration.

At the infamous Santa Cruz meeting in March of 1996 the issue was raised again. The text below is from version 15 of my template issues paper and covers the discussion in Santa Cruz.

> 3.28 *Elaborated-type-specifier*s in function template declarations revisited.
>
> In Waterloo, we decided that an *elaborated-type-specifier* containing a template parameter name could not be used in a function template declaration. Now that we have the partial ordering rules for function templates, this issue should be checked to see if it is still what we want. With the partial ordering rules, we can now select one template over another based on one being "more specialized" than another. It seems that these rules could be applied to elaborated type specifiers as well. If this is permitted in the partial ordering of function templates, it should also be permitted in the partial ordering used for class template partial specializations.

```
template <class T> class List {};
template <class T> void f(List<struct T> l){} // #1
template <class T> void f(List<union T> l){}  // #2
template <class T> void f(List<enum T> l){}   // #3
template <class T> void f(List<T> l){}        // #4

union U {};
struct S {};
class C {};
enum E {};

int main() {
      List<U> u;
      List<S> s;
      List<C> c;
      List<E> e;
      List<int> i;

      f(u); // calls #2
      f(s); // calls #1
      f(c); // calls #1
      f(e); // calls #3
      f(i); // calls #4
}
```

> Answer: Core-3 decided that the current rule banning use of template parameters in *elaborated-type-specifier*s only in function template declarations was not sufficient (because of things like partial specializations of classes), and that a simple prohibition against the use of template parameters in *elaborated-type-specifier*s was more desirable than a more complicated rule.

III. Proposed Resolution

The rationale for the restrictions on *elaborated-type-specifier*s (for template type parameters, at least) still seems valid. As noted in the introduction above, however, it is the interaction of these restrictions with the requirement that `friend` class declarations must use *elaborated-type-specifier*s that causes the problem. This paper therefore proposes to eliminate the latter requirement by permitting two additional syntactic non-terminals to be declared as `friend`s. The first is *simple-type-specifier*, whose definition is

simple-type-specifier:
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *type-name*
`::`$_{opt}$ *nested-name-specifier* `template` *template-id*
`char`
`wchar_t`
`bool`
`short`
`int`
`long`
`signed`
`unsigned`
`float`
`double`
`void`

type-name:
class-name
enum-name
typedef-name

The second non-terminal comes from the paper "Consolidated edits for core issues 245, 254, et al." (J16/02-0034 = WG21 N1376) by Clark Nelson, which was adopted as a defect report for inclusion in the Working Paper in April, 2003:

typename-specifier:
`typename ::`$_{opt}$ *nested-name-specifier identifier*
`typename ::`$_{opt}$ *nested-name-specifier* `template`$_{opt}$ *template-id*

With these additions, the problematic declarations described in the introduction could be reformulated as follows:

```
template <typename T>
class no_children {
      no_children () { }
      friend T;
};

template <class Allocator>
class X {
      friend typename Allocator::template rebind<X>::other;
};
```

```
template <class T>
class Y {
      friend typename T::U;
};
```

This change is a pure extension, in that it changes the meaning of no programs that are currently well-formed. It has very limited interaction with the rest of the Standard, because the syntax is already permitted – the syntax for a *member-declaration* is:

member-declaration:
decl-specifier-seq$_{opt}$ *member-declarator-list*$_{opt}$ `;`
`. . .`

Thus a `friend` keyword followed by any kind of *type-specifier* is simply a *member-declaration* with an omitted *member-declarator-list*. It is only explicit constraints in the text of the Standard that cause the reformulated declarations above to be ill-formed.

IV. Non-class `friend`s

There is no benefit to declaring a non-class type as a `friend`; no type other than a class can utilize the increased access privileges such a declaration affords. This fact is implicit in the current restriction requiring non-function `friend` declarations to use *elaborated-type-specifier*s. (Although an `enum` can be named in an *elaborated-type-specifier*, such an *elaborated-type-specifier* can only refer to a completely-defined `enum`; it cannot be used to introduce an incomplete type. Thus a hypothetical `friend enum` declaration would come too late: the `enum` would already have been defined and could not use any members of the class containing the `friend` declaration.)

The syntax proposed herein, however, opens at least the theoretical possibility declaring non-class types as `friend`s. The question of what restrictions should apply, if any, to the types named in `friend` declarations was vigorously discussed on the Evolution WG email reflector.

The first question to answer deals with `friend` declarations in class templates where the type specifier names a dependent type (14.6.2.1). In such cases, it is only upon instantiation of the template that the determination can be made whether the type in the `friend` declaration is a class or not. In its simplest form, the question is whether

```
template <typename T> class X {
      friend T;
};

X<int> xi;
```

is well-formed or ill-formed because of the attempt to declare `int` to be a `friend` of `X<int>`.

In considering this question, it is helpful to note that a `friend` declaration does not require anything of the befriended type – it simply offers that type the opportunity to access private and

protected members of the befriending class, and that grant of access need not be used. In this light, it seems unreasonable that the instantiation of a class template should fail, simply because the type offered this extra access is incapable of using it.

There was no disagreement on this point from any of the participants in the email discussion. Differing opinions were expressed, however, regarding the other contexts in which a non-class `friend` declaration might appear: non-template classes, and non-dependent types inside class templates. There were three distinct positions taken by various participants:

1. There should be no restrictions on the types named in `friend` declarations.

2. Except for `friend` declarations instantiated from dependent types, only class types (possibly cv-qualified) should be permitted as `friend`s.

3. Fundamental types named by their built-in keywords (`int`, `bool`, etc.) should be errors, but any type named by a user-declared name should be permitted.

The rationale for the first position is based on simplicity and consistency. Non-class `friend` declarations are harmless, even if useless. Because such declarations must be accepted when they arise via dependent types, the simplest and most consistent rule would be to accept them in all contexts, rather than having a special case for template code.

Supporters of the second position observe that, in the non-template and non-dependent cases, the programmer explicitly names the type and thus knows (or should know) what the type is. Because a non-class `friend` is useless, declaring one can only be the result of a mistake on the programmer's part – either he/she thought it did have an effect, or he/she specified a different name from the one intended – and we should require that it be diagnosed.

The third position is intermediate between the first two. There is never a reason for a programmer to declare a fundamental type to be a `friend` using its built-in keyword, so any such attempt should be diagnosed as an error. On the other hand, specifying a user-declared name as the type is less clearly an error. For instance, one could conceive of a situation where a given type was originally a smart pointer but in the course of program evolution it was replaced by a built-in pointer. It is unclear that such a change should cause a preexisting `friend` declaration to become an error.

Although the author of this paper favors the second position, he acknowledges that the arguments in favor of the other possibilities have some merit. As a result, the next section presents three possibilities for wording changes to the Standard and makes no recommendation as to the ultimate outcome.

V. Suggested Wording

As noted above, the syntax presented in this proposal is already accepted by the current grammar; the constraints that require an *elaborated-type-specifier* occur in only two locations in the Standard. The first of these is found in 9.2¶7:

> The *member-declarator-list* can be omitted only after a *class-specifier*, an *enum-specifier*, or a *decl-specifier-seq* of the form `friend` *elaborated-type-specifier*.

Rather than expanding the description of the acceptable syntax of `friend` declarations to include the additional productions, it would be better to have that specification at a single point in the Standard:

> The *member-declarator-list* can be omitted only after a *class-specifier* or an *enum-specifier* or in a `friend` declaration (11.4).

The second constraint occurs in 11.4¶2:

> An *elaborated-type-specifier* shall be used in a friend declaration for a class. *[Footnote:* The *class-key* of the *elaborated-type-specifier* is required.*]*

This wording should be deleted, and the following should be added as a new paragraph following 11.4¶2:

> A `friend` declaration that does not declare a function shall have one of the following forms:
>
> `friend` *elaborated-type-specifier* `;`
> `friend` *simple-type-specifier* `;`
> `friend` *typename-specifier* `;`
>
> ***[Insert wording here describing handling of non-class type specifiers: see below]*** [*Example:*
>
> ```
> class C;
> typedef C Ct;
>
> class X1 {
>       friend C;          // OK: class C is a friend
> };
>
> class X2 {
>       friend Ct;         // OK: class C is a friend
>       friend D;          //error: no type-name D in scope
> };
>
> template <typename T> class R {
>       friend T;
> };
>
> R<C> rc;                 // class C is a friend of R<C>
> R<int> ri;               // OK: "friend int;" is ignored
> ```
>
> *—end example*]

As noted earlier, this paper takes no position on which, if any, non-class types should be allowed in non-dependent `friend` declarations. The following wording (to be inserted at the location indicated above) implements the respective option:

Option 1:

A `friend` declaration whose type specifier designates a non-class type is ignored.

Option 2:

The type specifier in a `friend` declaration shall either designate a (possibly cv-qualified) class type or be a dependent type (14.6.2.1). If a class template is instantiated such that a dependent type results in a `friend` declaration that designates a non-class type, the `friend` declaration is ignored.

Option 3:

In a declaration of the form `friend` *simple-type-specifier*, the *simple-type-specifier* shall specify a previously-declared user-defined type or *typedef-name* (7.1.5.2). A `friend` declaration whose type specifier designates a non-class type is ignored.

VI. An Alternative Formulation

The suggested wording changes in the preceding section were intended to be compatible with all three positions regarding the treatment of non-dependent type specifiers, and they leave the current grammar intact. If a consensus develops around the third option, there is an alternative set of changes that might be simpler and more attractive.

The suggested wording for Option 3 verbally excludes the fundamental type keywords that are part of the *simple-type-specifier* nonterminal (7.1.5.2¶1). Another possibility would be to refactor that nonterminal into user-declared and built-in types, as follows:

simple-type-specifier:
user-type-specifier
type-key

user-type-specifier:
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *type-name*
`::`$_{opt}$ *nested-name-specifier* `template` *template-id*

type-key:
`char`
`wchar_t`
`bool`
`short`
`int`
`long`
`signed`
`unsigned`

```
float
double
void
```

With this refactorization, the suggested wording to implement Option 3 becomes:

> A `friend` declaration that does not declare a function shall have one of the following forms:
>
> `friend` *elaborated-type-specifier* `;`
> `friend` *user-type-specifier* `;`
> `friend` *typename-specifier* `;`
>
> A `friend` declaration whose type specifier designates a non-class type is ignored. [*Example:*
>
> `. . .`