prohibit friends of local classes ?

X3J16/92-0119
WG21/N0196

# Friend Declarations in Local Classes

R. Michael Anderson
Edison Design Group, Inc.
uunet!edg1!rma or rma@edg.com

October 30, 1992

This paper explores issues surrounding the declaration of friend functions and friend classes in a local class. It demonstrates that it's pointless for a local class to grant friendship to a function unless the function is also defined locally, but that defining a file-scope function[1] inline at the point of the local-class friend declaration may introduce problems, since it can also be called nonlocally. To avoid these problems friend declarations in a local class could either be prohibited altogether or be allowed only with restrictions. It is recommended that *friend declarations in a local class be disallowed except to grant friendship to other classes local to the same function and to member functions of such classes.*

## The need for local definition

When a class or function is defined at file scope and is befriended by a local class, the local-class friend declaration is completely useless, since the befriending class is invisible in the scope in which the befriended function or class is defined. Consider this example:

```
void f1();
class G1 { void f2(); };
class G2;
void foo()
{
  class L1;
  class L2 {
    friend void f1();              // file-scope function
    friend void G1::f2();          // member of file-scope class
    friend class G2;               // file-scope class
    friend class L1;               // local class
  };
  class L1 { /* ... */ };          // L2 is visible
}
void f1() { /* ... */ }            // L2 is invisible
void G1::f2() { /* ... */ }        // L2 is invisible
class G2 { /* ... */ };            // L2 is invisible
```

The friendship conferred by local class L2 is wasted on the file-scope entities `f1()`, `G1::f2()`, and G2, since they have no way to refer to the nonpublic members of L2 to which they are given access. Only L1 can make use of the friend declaration.[2]

[1] I use the term "file-scope function" to refer to any nonmember function and to member functions of a file-scope class. Similarly, a "file-scope class" is any (nested or nonnested) class that is not a local class, and a "local-scope function" is a member function of a (nested or nonnested) local class.

[2] Note that this would not have been the case if L1's definition had preceded that of L2; we *could* relax the rules to allow the definition of a local-class member function to be delayed and appear in a friend declaration.

For a local-class friend declaration of a file-scope function to be *useful*, that function needs to be defined locally (typically, as an inline definition accompanying the friend declaration).[3] Thus, the previous example could be rewritten to allow the file-scope functions to take advantage of the friend declarations:

```
void f1();
class G { void f2(); };
void foo()
{
  class L {
    friend void f1() { /* ... */ }
    friend void G::f2() { /* ... */ }
  };
}
```

Now, since they are defined locally, `f1()` and `G::f2()` are in a position to exploit the friendship conferred by class `L` and to refer to the latter's nonpublic members.[4]

## Problems with local definition

Like member functions of local classes, file-scope friend functions that are defined locally have partial access to local data. I say this assuming that the language in WP 9.8 ¶1 applies to friend declarations as well as to member function declarations: "declarations in a local class can use ...static [but not automatic] variables ...."[5] Thus the following program is legal:

```
static int f();
void foo()
{
  static int j = 0;
  class L {
    friend int f() { ++j; }
  };
}
```

The problem here is that there is no reason why file-scope function `f()` could not be called from outside `foo()`, even though it has special access to data and functionality that would otherwise be hidden. I consider this a problem for two reasons:

- *It turns functions into quasi-modules.* In effect, there would be multiple entry points into the environment defined by the function. This might permit some novel C++ programming, but it would not be consistent with what I understand to be "the spirit" of the language.[6]

[3] There doesn't appear to be any way in which a local-class friend declaration of a file-scope class can be made useful.

[4] Should the language disallow local-class friend declarations of file-scope classes and of file-scope functions that are not defined locally? The uselessness of a feature is no reason to outlaw it, but no one would be likely to miss this one if it were prohibited.

[5] Local-class friend function declarations are obliquely referred to in WP 9.7 as well, where another similarity of friend functions to member functions is affirmed: "like a member function, a friend function defined within a class ... has no special access rights to ... local variables of an enclosing function".

[6] One could argue that being able to pass around the address of a static member function of a local class also creates the possibility of writing a function with multiple entry points.

- *It permits access to uninitialized local variables.* This can be viewed as a "safety" issue. Consider this example:

```
static int f();
int x = f();
int y = -1;
void g()
{
  static int i = y;
  class L {
    friend int f() { return i; }
  };
}
```

What is the value returned by `f()` to initialize `x`? The wording of WP 6.7 ¶4 says that `i` has the value `0` before the first entry into the routine block for `g`, but I'm not sure that's the same as saying it has that value when the initialization of `x` takes place. (Does "before" mean "any time before" or "just before"?) The real issue is whether it should be permissible to access a local static variable (maybe even to modify it) before the function to which it belongs is called (if it is called at all).

## Some solutions

Thus there are two serious (though perhaps not fatal) problems with locally defined file-scope functions. These problems can be alleviated by imposing restrictions on local-class friend declarations. Here are some possibilities.[7]

1. *Prohibit all references to local data from file-scope functions.*

   If this restriction were imposed, the above example would be illegal because not even static data members could be referenced by a file-scope friend function. This would provide a solution to the second problem, the possibility of accessing uninitialized local static variables. It wouldn't do much about the first problem, though: it would still be possible to write module-like functions with multiple entries.

2. *Prohibit nonlocal references to locally defined file-scope functions.*

   A local-class friend declaration of a file-scope function would be permitted: it could be defined locally and given the same access to local data that a local-class member function has. However, it could not be called from outside the function in which its definition appeared (nor could its address be taken nonlocally).

   The previous example, then, would be illegal because `f()` is called outside the scope of `g()`. Even though `f()` belongs to the file scope and is visible outside the local scope, it behaves like a nested function and can only be called as though it were.

   This would work, but I find it an unattractive solution—mainly because it is counterintuitive (or at least bizarre) to have file-scope functions that are really sort of like nested functions but still intrude into the file-scope name space but can't be called, etc.

[7] This is not an exhaustive sample. For example, the concept of "local nonmember functions" could be introduced into the language, i.e., nonmember functions visible only in the scope in which they are declared; they could be defined to distinguish them from "true nested functions."

3. *Prohibit inline definitions of file-scope functions.*

   Although this would be a straightforward solution, easy to describe and justify, easy for users to understand, and (for what it's worth) easy for compilers to implement, it would amount to eliminating local-class friend declarations of file-scope functions, since without an inline definition they would be useless.

   If this option were adopted, only other local classes and their member functions could be befriended by a given local class. But that might well be sufficient: I don't believe that much functionality (if any) would be given up by excluding file-scope functions from local-class friend declarations.[8]

## Recommendation

These, to summarize, seem to me to be the set of plausible approaches to local-class friend declarations:

A. *Disallow local-class friend declarations altogether.* This Gordian-knot solution is appropriate only if the feature is not *really* a useful and desirable part of the language.

B. *Disallow local definitions of nonlocal functions.* This would effectively limit *useful* local-class friend declarations to declarations of other local classes and of local-class member functions.

C. *Prohibit locally defined file-scope functions from referring to local variables.* That is, allow "safe" local definitions of nonlocal functions.

D. *Prohibit nonlocal references to locally defined file-scope functions.* That is, eliminate nonlocal access to potentially unsafe functions.

E. *Impose no additional restrictions on local-class friend declarations.* It would then be a quality-of-implementation issue for compilers to warn about unsafe uses of this feature. We might also want to improve the language about the "preinitialized" values of local static variables.

As a compiler-writer I am biased towards a solution that is straightforward, easy to describe and implement, and consistent with (what I take to be) "the spirit of C++." This inclines me towards B, the effect of which is summarized by either of the following rules:

- *Friend functions declared in a local class may not be defined within the class definition.*
- *A friend declaration in a local class is allowed only when it names either another local class (one belonging to the same lexical scope or a scope contained by it) or a member function of such a class.*

Approach A seems a bit too Draconian, and the others are unattractive because file-scope functions that can make use of local-scope functionality and data seem out of place in the language, even if there are ways to make them work safely.

That's my judgment as an implementor. It would be interesting to know what experienced users of C++ have found. Are file-scope functions ever used as friends of local classes? Could one do without them? Is friendship between local classes ever used?

[8] A case might be made for member functions of a file-scope class: such a function, defined in the context of the local class but also a member of the file-scope class, would have access to private members in both contexts. But I don't think it would be difficult to find alternatives to duplicate this functionality.