Doc no.: J16/00-0045 = WG21 N1268
Author: Alan Griffiths,
Mark Radford,
email: alan@octopull.demon.co.uk
mark@twonine.demon.co.uk
Date: 17-October-2000

Member Access Control and Nested Classes

Preamble

Discussions within IST/5/-21 of Martin J. O'Riordan's proposal "Member Access Control - Proposed Revisions" (WG21 N1254/J16 00-0031) [N1254] identified significant reservations by some members of the committee (including the authors).

These reservations are equally applicable to the proposed resolution of core-45. (In addition to this we are in agreement with Mr O'Riordan's comments regarding the use of the friend mechanism and the limited scope of the proposed solution.)

The substance of our reservations is that it involves a change from the access rights for nested classes that are both widely implemented and widely known and used within the C++ community. From the perspective of a user of C++ the benefits of the change are unclear and the change would 'break' a number of popular idioms.

It became apparent that an alternative proposal that attempts to address these concerns should also be considered.

Changes

Accept the proposed resolution to core-77. This is repeated here for convenience:

1. In 11.4 class.friend paragraph 1, change

   > A friend of a class is a function or class that is not a member of the class but is permitted to use the private and protected member names from the class. The name of a friend is not in the scope of the class, and the friend is not called with the member access operators (5.2.5 expr.ref) unless it is a member of another class.

   to

   > A class may give access to its private and protected members to functions and classes that might not otherwise have access by making those functions and classes *friends*. Friendship is indicated by a `friend` declaration within the class, but such a declaration does not introduce a member.

   Also, following the example, add the sentence:

   > A class that is a member of another class does not gain any special access to the enclosing class. However, such member classes may be declared as friends of the enclosing class.

2. In 11.4 class.friend paragraph 2, add the following lines to the second example:

```
class W {
    class X;          // declare a member class,
distinct from ::X
    friend class X;  // make it a friend
    class X {         // define the member class
        int var;
    };
};
```

Note that without the initial declaration of `W::X`, the friend declaration would declare `::X` as a friend of `::W`.

In addition, accept the proposed resolution to core-209. This is repeated here for convenience:

3. remove the first sentence of 11.4 class.friend, paragraph 7:

   A name nominated by a friend declaration shall be accessible in the scope of the class containing the friend declaration. The meaning of the friend declaration is the same whether the friend declaration appears in the private, protected or public (class.mem) portion of the class member-specification.

   which becomes:

   The meaning of the friend declaration is the same whether the friend declaration appears in the private, protected or public (class.mem) portion of the class member-specification.

Also, for consistency:

4. In 11 class.access paragraph 3, change:

   Access control is applied uniformly to all names, whether the names are referred to from declarations or expressions. [Note: access control applies to names nominated by friend declarations (class.friend) and using-declarations (namespace.udecl).] In the case of overloaded function names, access control is applied to the function selected by overload resolution.

   to

   Access control is applied to all names, whether the names are referred to from declarations or expressions with the exception of friend declarations (class.friend). [Note: access control applies to names nominated by using-declarations (namespace.udecl).] In the case of overloaded function names, access control is applied to the function selected by overload resolution.

Finally, there is significant merit to the idea (proposed in N1254) of extending the utility of local classes:

5. In 11 class.access paragraph 1, insert the following paragraph:

   A local class of a member function may access the same names that the member function itself may access.

NB This last change is not key to the arguments made in this paper.

Discussion

A number of examples are cited in N1254 and were raised during the IST/5/-21 discussions. These are not repeated where they follow N1254. The following identify the effect of the above changes when compared with that paper.

Example 1

The following example from N1254:

```
class X {
void acquire ();
void release ();
class Manage {
      X& rX;
      public:
      Manage ( X& r ) : rX(r) { rX.acquire (); }
      ~Manage () { rX.release (); }
      };
public:
T use ();
};
```

with the above changes this design can be rendered:

```
class X {
void acquire ();
void release ();
class Manage {
      X& rX;
      public:
      Manage ( X& r ) : rX(r) { rX.acquire (); }
      ~Manage () { rX.release (); }
      };
friend Manage;
public:
T use ();
};
```

Example 2

The following example from N1254:

```
class X;

class Resource {
void acquire ();
void release ();
friend class X;
};

class X {
Resource res;
class Manage {
      X& rX;
      public:
      Manage ( Resource& r ) : rX(r) { rX.acquire (); }
      ~Manage () { rX.release (); }
};
public:
T use ();
};
```

with the above changes this design can be rendered:

```
      class X;

      class Resource {
      void acquire ();
      void release ();
      friend class X;
      };

      class X {
      Resource res;
      inline void aquire_res() { res.acquire(); }
      inline void release_res() { res.release(); }
      class Manage;
      friend Manage;
      class Manage {
            X& rX;
            public:
            Manage ( Resource& r ) : rX(r) { rX.aquire_res(); }
            ~Manage () { rX.release_res(); }
      };
      public:
      T use ();
      };
```

Example 3

The following example was raised in discussion:

```
      class interface {
             . . .
      private:
            class implementation;
            implementation* grin;
      };
```

This is a common idiom (known variously as “Cheshire Cat”, “Compilation Firewall” and “pimpl”). It defines two classes ‘interface’ and ‘interface::implementation’. It is clear that the use of nested classes reflects the close coupling of the two classes.

That the above classes have controlled access to each other is well known *and is intended* by many developers.

The changes proposed by N1254 would remove the control of access to ‘interface’ by ‘interface::implementation’. The authors consider it highly undesirable to change the meaning of a large body existing code in this way.

For a developer to restore the desired access control requires a rewrite:

```
class implementation;
class interface {
       . . .
private:
      implementation* grin;
};
```

Which has the disadvantage of making the type ‘implementation’ accessible to consumer code.

Example 4

The following example was raised in discussion:

```
class A {
public:
      class X { . . . };
      class Y { . . . };
      . . .
};
```

Once again the changes proposed by N1254 would remove the control of access between classes that exists currently. Again it must be presumed that developers intend access to be controlled.

However, the relationship between classes is weaker than in example 3, a reasonable alternative design could place the currently nested classes into the surrounding namespace. (Admitttedly in some contexts this would need to be supplemented by typedefs within class A.)

Example 5

The following example was raised in discussion:

```
class Outer {
      class Wibble { friend Wobble::Woo; . . . };
      class Wobble { class Woo {…}; . . . };
};
```

With the suggested changes this can now be rendered:

```
class Outer {
      class Wobble { class Woo {…}; . . . };
      class Wibble { friend Wobble::Woo; . . . };
};
```

Conclusions

The overwhelming majority of nested classes in existing code accessible to the authors (and by their accounts other members of the BSI Committee) do not require access to the surrounding classes.

The fact that the standard does not permit access to be granted to nested classes does not impact the sample as the compilers being used implement a non-standard extension allowing nested classes to be declared friends, and the developers were unaware until recently that this is non-standard.

Controlling access is an important technique for reducing the maintenance costs of code. Removing it from the language entirely would not break any code – clearly existing code would still compiles with the same behavior. However, the development community would not welcome such a change – those consulted by the authors were almost uniform in their reaction: that being unable to declare nested classes “friend” would be a minor inconvenience whereas removing access control was unreasonable.

The fact that nested classes do not (by default) have access to private and protected members of the surrounding class is known and is used by developers to enhance the maintainability of their code. The idioms that exploit this will be broken by N1254. (Note a fine distinction is drawn between breaking the idiom, and breaking the code.)

The following issues cited in N1254 are addressed in the following way:

Related issues

[OPEN] CWG Issue #8, sub Issue 2

This is not a defect.

(In the following I assume from the text that follows that the friend declaration in the example should read “`friend int_temp<A1, 5, func>;`”. Even so, the discussion under issue 1 is confused – the scope in question is that of `A::funct2`, not that of `A::int_temp<>`.)

In 11 paragraph 6 we have:

> “In the definition of a member of a nested class that appears outside of its class definition, the name of the member may be qualified by the names of enclosing classes of the member's class even if these names are private members of their enclosing classes.”

This clearly states that the qualified name of the member class (in the example an explicit full specialization of a template) may be used.

What it does not explicitly address is the use of the template parameters that identify this specialization. Since the explicit specialization within the body has already introduced the class name the accessibility of the template parameters at this point should be irrelevant. (They are part of a composite entity that we are permitted to use.)

[REVIEW] CWG Issue #9

Not affected, the existing proposed resolution is still good and necessary.

[OPEN] CWG Issue #10

This is still an issue. Some possibilities for resolving it are discussed below.

The language is inconsistent in that it uses two distinct scopes in interpreting a member definition at namespace scope. Names are resolved from namespace scope, while access is checked from the scope of the member’s class.

The following example is adapted from the issue text (there are several examples given but the arguments apply to all of them). Given the class:

```
class D {
    class E {
        static E* m;
        };
    };
```

Is the following member definition well formed? (With specific reference to `D::E*`)

```
 D::E* D::E::m = 0;
```

In 11 paragraph 5 we also have the following:

> "The access control for names used in the definition of a class member that appears outside of the member's class definition is done as if the entire member definition appeared in the scope of the member's class."

From this the access controls for a definition outside the class should be assessed identically to those for a declaration inside the class. That is as though the example were rewritten:

```
    class D {
        class E {
            static D::E* m;
        };
    };
```

Which is similar in all important respects to the example given in 11.8 paragraph 2 (which states that `D::E` is inaccessible).

A tentative resolution

In 11 paragraph 6 replace:

> "In the definition of a member of a nested class that appears outside of its class definition, the name of the member may be qualified by the names of enclosing classes of the member's class even if these names are private members of their enclosing classes."

With:

> "In the definition of a member of a nested class that appears outside of its class definition, the name of its class may be qualified by the names of enclosing classes even if these names, or the name of its class, are private members of their enclosing classes."

Note however, that this does not resolve all possible inconsistencies. A more sweeping rewording resolves this, but may have other implications:

> "In the definition of a member of a nested class that appears outside of its class definition, names that would be accessible within the class scope may be further qualified by the names of enclosing classes even if any, or all, of these names are private members of their enclosing classes."

[REVIEW] CWG Issue #16

Not affected, the existing proposed resolution is still good and necessary.

[REVIEW] CWG Issue #45

This is not a defect.

- 10 paragraph 1 is clear in specifying the requirements placed on the legality of a base class name for the program to be well formed. The legality of using Parent in the base-clause has not been questioned.

- Also in 10 paragraph 1 it is stated “Inherited members can be referred to in expressions in the same manner as other members of the derived class, unless their names are hidden or ambiguous (class.member.lookup).”

- Members of Derived can therefore reference members of the Parent without referencing names in the scope of C.

- 11.2 makes clear the access to base class members afforded to the derived class.

It follows that (with the exception of the non-standard header) the example code is well formed.

[READY] CWG Issue #77

Not affected, the existing proposed resolution is adopted as part of this proposal.

[READY] CWG Issue #142

Not affected, the existing proposed resolution is still good and necessary.

[DR] CWG Issue #161

Not affected, the existing proposed resolution is still good and necessary.

[DRAFTING] CWG Issue #207

Not affected, the existing proposed resolution is still good and necessary.

[READY] CWG Issue #209

Not affected, the existing proposed resolution is adopted as part of this proposal.