Doc. No.: WG21 N1254/J16 00-0031
Author: Martin J. O'Riordan
EMail: martino@theheart.ie
Date: 6 September 2000

Member Access Control - Proposed Revisions

1 Changes

After 11¶1, insert the following paragraph and add the footnote,

> A member of a class can also access all names as the class of which it is a member. A local class of a member function may access the same names that the member function itself may access (footnote).
>
> Footnote: Access permissions are thus transitive and cumulative to nested and local classes.

Delete the whole of 11¶6, as it is no longer irrelevant.

Replace the first sentence of Section 11.8¶1,

> The members of a nested class have no special access to members of an enclosing class, nor to classes or functions that have granted friendship to an enclosing class; the usual access rules (clause 11) shall be obeyed.

With the following,

> A nested class is a member and as such has the same access rights as any other member.

Strictly speaking, just deleting the first sentence of 11.8¶1 is all that is necessary, but replacing it with the above words makes it clear that the intention is that a nested class is a member like any other.

In the example 11.8¶1, replace the lines that read,

```
B b;       // error: E::B is private
```

and,

```
p->x = i;  // error: E::x is private
```

With the following,

```
B b;       // Okay, E::I can access E::B
```

and,

```
p->x = i;  // Okay, E::I can access E::x
```

respectively.

Delete 11.8¶2 altogether.

2 Discussion and Examples

There are many issues concerning accessibility and nested classes that are inconsistent with other kinds of members, especially member functions.

A class describes two main parts:

?? an interface

?? an implementation

Usual thinking is to consider that the consumer of the class may use the interface to the class without having any concern about its implementation. Indeed, the implementation should be freely constructed after the fashion best determined by the implementor, and remain opaque to the consumer of the interface.

I have often thought that the implementation of a member function should be no more entitled to privilege than a member class and vice versa. But this is not that case in the current IS. All of the block scopes in a member function have the complete member access rights, which currently the member class does not enjoy.

2.1 Local Classes and Member Access Control:

I was considering an example recently,

```
class X
{
private:
static  bool inside_foo;
public:
        T    foo ();
};

bool X::inside_foo = false;

T X::foo ()
{
    inside_foo = true;
    // some code
    if ( some-conditional-expression )
    {
        // some more code
        inside_foo = false;
        return value;
    }
    else if ( some-other-conditional-expression )
    {
        // other code
        inside_foo = false;
        return value;
    }
    // yet more code
    inside_foo = false;
    return value;
}
```

Here I have had to be very careful to remember to reset the '`inside_foo`' boolean on each return path. This leads to a lot or potential errors as the function is edited, and return paths may be missed. Obviously the example is concocted, but it is nevertheless illustrative of general resource management issues. Typically, such code might be rewritten such that another class does the work of managing the resource, for example,

```
T X::foo () {
    class Helper { public:
            Helper ()  { X::inside_foo = true;  }
            ~Helper () { X::inside_foo = false; }
    } helper;
    // some code
    if ( some-conditional-expression )
    {   // some more code
        return value;
    }
    else if ( some-other-conditional-expression )
    {   // other code
        return value;
    }
    // yet more code
    return value;
}
```

And the simplicity and robustness of the code is improved. However, the current IS does not support this use as the local class doesn't have any access to '`X::inside_foo`'. Yet doing so is strictly an implementation choice that has no effect on the consumer of the `class X`, nor does it in anyway compromise the access security of the `class X`.

[Note: This particular example could be coded by adding a reference to '`int`' and binding '`inside_foo`' to it when declaring the object '`helper`']

In this case, the proposed resolution that a nested class is implicitly made a '`friend`' of the class of which it is a member could be extended to local classes. However, I have several problems with this, two of which are that,

?? the idea of expressing the accessibility of members in terms of `friend`'s is semantically broken. It should always work that '`friend`' access is described in terms of member access, not the other way around. It is '`friend`' that represents the special case, not members.

?? the implicit '`friend`' hack only goes to one level of nesting. If the nested class is to be granted access rights to its sibling members only through friendship, it means that the implementation choices for the nested class itself are restricted. The implementor cannot describe the nested class implementation in terms of sub-nested classes because friendship is not transitive.

The implicit `friend` hack does increase the right of the implementor to choose a nested class for their implementation over a function style implementation, for example,

```
class X {
    void     acquire ();
    void     release ();
public:
    T        use ();
};

T X::use () {
    acquire ();
    if ( conditional-expression )
    {   release ();
        return value;
    }
    release ();
    return anotherValue;
}
```

With the implicit `friend` hack, the above example can be rewritten as,

```
class X {
    void     acquire ();
    void     release ();

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

T X::use () {
    Manage mgr (*this);
    if ( conditional-expression )
        return value;
    return anotherValue;
}
```

The implementation is simplified and made more robust. As with the previous local class example, this poses no threat to the security or interface of the `class X` by its consumers, but having the implicit `friend` hack does increase the choices of implementation for the implementor.

However, even a slight variation of the implicit `friend` hack does not work. Consider the following example,

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

class X {
    Resource res;
public:
    T        use ();
};

T X::use () {
    res.acquire ();
    if ( conditional-expression )
    {   res.release ();
        return value;
    }
    res.release ();
    return anotherValue;
}
```

In this case, the implicit `friend` hack does not permit the implementor to make the equivalent implementation choice by using a nested class. That is, the following analogous code is not permitted,

```
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 ();
};

T X::use () {
    Manage mgr (res);
    if ( conditional-expression )
        return value;
    return anotherValue;
}
```

And yet, nothing has really changed. In this case, `class Resource` has stated that the `class X` has special access. It has not placed an implementation constraint on the `class X`. Yet, the implementation of `class X` is implicitly restricted to NOT using a nested class, despite the fact that doing so compromises neither the security nor interface of either the `class Resource` or the `class X`.

The proposed implicit `friend` mechanism only goes partway. Indeed, it becomes more ludicrous when the nested class has itself a sub-nested class intended for similar purpose,

```
class Something {
    class X;
    class Resource {
        void     acquire ();
        void     release ();
    friend class X;
    };
    class X {
        Resource res;
    public:
        T        use ();
    };
};

T Something::X::use () {
    res.acquire ();
    if ( conditional-expression )
    {   res.release ();
        return value;
    }
    res.release ();
    return anotherValue;
}
```

But it cannot be rewritten by the implementor as,

```
class Something {
    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 ();
    };
};

T Something::X::use () {
    Manage mgr (res);
    if ( conditional-expression )
        return value;
    return anotherValue;
}
```

Despite the fact that it is not really different at all to the global case. The security and interface of both the `class Something` and the `class Something::Resource` remains un-compromised, yet this implementation choice is unavailable.

3 Summary

Allowing access permissions of members to be inwardly transitive resolves quite a lot of the issues concerning member access control. It also seems to be the natural way to provide the needed access security, while at the same time not compromising the implementor in their choice of implementation strategy. It seems very strange that the simple resource management metaphor is restricted in the way it currently is. The implicit `friend` approach is only a half-hearted attempt at resolving this anomaly. The proposal in this document takes that partial solution, and extends it completely to address the anomalies that implicit `friend` was intended to resolve by generalising it to transitive accessibility for members.

3.1 Reconsideration of other Member Access Control issues:

Looking at the other issues concerning Clause 11 in the light of the above proposal, it seems that with this one semantic change, quite a lot of the issues are resolved; even those for which we have other special case resolutions already.

This proposal directly resolves issues #8, #10 and #45, and partially resolves issue #77 which would require simple revision. I also expect that it resolves many other anomalies in access control that have come up from time to time.

For completeness I have examined each of the issues and summarised the affects of this proposal on them.

3.1.1 [OPEN] CWG Issue #8, sub Issue 2:

The changes to 11.8 make this work as expected. The usual wording of 11¶5 is all that is subsequently necessary to check accessibility.

This issue is resolved by the proposed changes in this paper.

3.1.2 [REVIEW] CWG Issue #9:

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

3.1.3 [REVIEW] CWG Issue #16:

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

3.1.4 [READY] CWG Issue #142:

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

3.1.5 [DRAFTING] CWG Issue #207:

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

3.1.6 [READY] CWG Issue #77:

The proposed resolution would need to be revised. In the currently proposed changes to 11.4¶1, remove the words,

> 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.

Also, the second part of the proposal (to 11.4¶2) should be removed completely, as it is no longer relevant.

3.1.7 [READY] CWG Issue #209:

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

3.1.8 [DR] CWG Issue #161:

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

3.1.9 [OPEN] CWG Issue #10:

This issue is resolved by the proposed changes in this paper. Because of transitive access for nested members, the words of 11¶5 are sufficient to resolve this.

This issue is resolved by the proposed changes in this paper.

3.1.10 [REVIEW] CWG Issue #45:

This issue is also resolved by the proposed changes in this paper. Again, the transitive accessibility properties for nested classes resolves this problem.

The existing proposed resolution for issue #45 is replaced by this proposal.