Why does the compiler refuse to let me export my class? If I don't export it, then the class works fine.

devblogs.microsoft.com/oldnewthing/20190927-00

September 27, 2019

Raymond Chen

Consider the following class:

```
class Node
{
public:
 Node();
 // ... public methods ...

private:
 std::vector<std::unique_ptr<Node>> children;
};
```

This class works great as long as you don't try to export it. Adding the `__declspec(dllexport)` declaration specifier causes the compiler to get mad.

```
class __declspec(dllexport) Node
{
public:
 Node();
 // ... public methods ...

private:
 std::vector<std::unique_ptr<Node>> children;
};
```

The Visual C++ compiler now freaks out with

```
xutility(1784): error C2280: 'std::unique_ptr<Node, std::default_delete<_Ty>>
&std::unique_ptr<_Ty, std::default_delete<_Ty>>::operator =(const
std::unique_ptr<_Ty, std::default_delete<_Ty>> &)': attempting to reference a deleted
function
        with
        [
            _Ty=Node
        ]
memory(1969): note: see declaration of 'std::unique_ptr<Node,
std::default_delete<_Ty>>::operator ='
        with
        [
            _Ty=Node
        ]
memory(1969): note: 'std::unique_ptr<Node, std::default_delete<_Ty>>
&std::unique_ptr<_Ty, std::default_delete<_Ty>>::operator =(const
std::unique_ptr<_Ty, std::default_delete<_Ty>> &)': function was explicitly deleted
        with
        [
            _Ty=Node
        ]
...
vector(1044): note: while compiling class template member function 'void
std::vector<std::unique_ptr<Node, std::default_delete<_Ty>>,
std::allocator<std::unique_ptr<_Ty, std::default_delete<_Ty>>>>::_Copy_assign(const
std::vector<std::unique_ptr<_Ty, std::default_delete<_Ty>>,
std::allocator<std::unique_ptr<_Ty, std::default_delete<_Ty>>> &, std::false_type)'
        with
        [
            _Ty=Node
        ]
vector(1062): note: see reference to function template instantiation 'void
std::vector<std::unique_ptr<Node, std::default_delete<_Ty>>,
std::allocator<std::unique_ptr<_Ty, std::default_delete<_Ty>>>>::_Copy_assign(const
std::vector<std::unique_ptr<_Ty, std::default_delete<_Ty>>,
std::allocator<std::unique_ptr<_Ty, std::default_delete<_Ty>>> &, std::false_type)'
being compiled
        with
        [
            _Ty=Node
        ]
node.h(11): note: see reference to class template instantiation
'std::vector<std::unique_ptr<Node, std::default_delete<_Ty>>,
std::allocator<std::unique_ptr<_Ty, std::default_delete<_Ty>>>>' being compiled
        with
        [
            _Ty=Node
        ]
```

Let's simplify the error messages by taking out the default template parameters and the method return values, and performing the substitutions.

```
xutility(1784): error C2280: 'std::unique_ptr<Node>::operator=(const
std::unique_ptr<Node> &)': attempting to reference a deleted function
memory(1969): note: see declaration of 'std::unique_ptr<Node>::operator='
memory(1969): note: 'std::unique_ptr<Node>::operator=(const std::unique_ptr<Node>
&)': function was explicitly deleted
...
vector(1044): note: while compiling class template member function 'void
std::vector<std::unique_ptr<Node>>::_Copy_assign(const
std::vector<std::unique_ptr<Node>> &, std::false_type)'
vector(1062): note: see reference to function template instantiation 'void
std::vector<std::unique_ptr<Node>>::_Copy_assign(const
std::vector<std::unique_ptr<Node>> &, std::false_type)' being compiled
node.h(11): note: see reference to class template instantiation
'std::vector<std::unique_ptr<Node>>' being compiled
```

What the compiler is trying to tell you is that it's instantiating the `std::vector` template with `std::unique_ptr<Node>` as the type parameter, and it ran into trouble with the copy assignment operator. In order to copy the vector, it needs to copy each element, but `std::unique_ptr<Node>` deleted its copy assignment operator.

Okay, that makes sense. You can't copy a vector of move-only objects. We saw that last time. But why does this problem occur only when you try to export the class? Shouldn't it always be a problem?

Template methods are instantiated on demand, and if you never try to copy the vector of move-only objects, then the copy assignment operator will never be invoked, and the compiler will never encounter the situation where it needs to copy a non-copyable object.

That piano is perfectly safe, as long as you don't play B4.

Exporting a class forces the compiler to generate code for *every method*, just in case some other module tries to call it. In particular, the compiler must generate the `Node` copy constructor and copy assignment operator, but it can't, and that's why you're getting the error.

Exporting the class forces the compiler to play every note on the piano, just in case somebody ever presses it.

To fix this problem, make your class explicitly non-copyable and non-copy-assignable.

```
class Node
{
public:
 Node();

 Node(Node const&) = delete;
 Node& operator=(Node const&) = delete;

 // ... public methods ...
private:
 std::vector<std::unique_ptr<Node>> children;
};
```

This removes the methods from the class, so the compiler won't try to export them. Those methods didn't work anyway, so removing them doesn't cause any harm. It also makes the error messages much more comprehensible.

```
void f(const Node& v)
{
  auto copy = v;
}
```

Instead of some horrible 50-line error message, you get a very direct explanation of the problem:

```
error C2280: 'Node::Node(const Node &)': attempting to reference a deleted function
note: see declaration of 'Node::Node'
note: 'Node::Node(const Node &)': function was explicitly deleted
```

Raymond Chen

Follow