

On resolving the type vs member conflict in C++: The Color Color problem

 devblogs.microsoft.com/oldnewthing/20190419-00

April 19, 2019



Raymond Chen

In C++, there are ambiguities when a member function has the same name as a type. Consider:

```
// Some header file from a UI library you are using

namespace Windows::UI::Xaml
{
    enum class Visibility { Collapsed, Visible };

    struct Style { /* ... */ };

    namespace Controls
    {
        struct UElement
        {
        public:
            /* ... */

            // returns current visibility
            Windows::UI::Xaml::Visibility Visibility();

            // change visibility
            void Visibility(Windows::UI::Xaml::Visibility value);

            // returns current style
            Windows::UI::Xaml::Style Style();

            // change style
            void Style(Windows::UI::Xaml::Style value);
        };
    }
}
```

Your project has a custom element, say like this:

```
namespace MyProject
{
    class CustomElement : public Windows::UI::Xaml::Controls::UIElement
    {
        void OnThemeChanged();
    };
}
```

And now you are implementing the `OnThemeChanged` method.

```
using namespace Windows::UI::Xaml;

void MyProject::CustomElement::OnThemeChanged()
{
    // Find out what style to use.
    Style style = GetStyleFromCurrentTheme();

    // Set it as our style.
    Style(style);
}
```

This code doesn't compile. The word `Style` can mean multiple things:

- It could refer to the class `Windows::UI::Xaml::Style`.
- It could refer to the method `MyProject::CustomElement::Style()`.

C++ unqualified name lookup in member functions searches the class before searching in namespaces, so it finds the method `CustomElement::Style()`. SFINAE does not apply, so if the name inside the class doesn't make sense, the compiler doesn't keep looking for a better match; it simply reports an error. Depending on how you used the name `Style`, the error message will vary, but whatever it is, it is usually incomprehensible.

```
error: expected ';' before 'style'
error: statement cannot resolve address of overloaded function

error: cannot determine which instance of overloaded function
'MyProject::CustomElement::Style' is intended
error: expected a ";"

error: syntax error: missing ';' before identifier 'style'
error: non-standard syntax; use '&' to create a pointer to member
```

These error messages make sense once you realize that the compiler resolved `Style` to the member function. It's a case of an error message written with compiler-colored glasses. To be fair, the compiler doesn't realize that it's wearing glasses. It's simply following the rules for unqualified name resolution and reporting the problems with the name you chose by mistake.

The only compiler I found that generates a helpful error is clang, which goes the extra mile and realizes that the name the language rules require it to choose may not have been the name you intended.

```
error: must use 'struct' tag to refer to type 'Style' in this scope
note: struct 'Style' is hidden by a non-type declaration of 'Style' here
    Windows::UI::Xaml::Style Style();
```

The clang compiler even provides a suggestion as to how you can force the name to be resolved the way you intended.

```
void MyProject::CustomElement::OnThemeChanged()
{
    // Find out what style to use.
    struct Style style = GetStyleFromCurrentTheme();

    // Set it as our style.
    Style(style);
}
```

Alternatively, you can use the scope resolution operator to force the name to be looked up in the global scope:

```
void MyProject::CustomElement::OnThemeChanged()
{
    // Find out what style to use.
    ::Style style = GetStyleFromCurrentTheme();

    // Set it as our style.
    Style(style);
}
```

The global lookup will work because you did a `using namespace Windows::UI::Xaml;` to import the names from that namespace into the global namespace. After all, that's what led you to believe that writing simply `Style` was good enough in the first place.

This problem also exists with the `Visibility` enumeration:

```
void MyProject::CustomElement::OnThemeChanged()
{
    // See if we should be visible in the current theme.
    Visibility visibility = IsVisibleInCurrentTheme();
    //^^^^^^^^^^ compiler error here

    // Set it as our new visibility.
    Visibility(visibility);
}
```

The solution is the same. You need to qualify the name in a way that prevents the compiler from considering the member function. You could say `enum Visibility` , or you could say `::Visibility` .

Note, however, that there is no problem here:

```
void MyProject::CustomElement::OnThemeChanged()  
{  
    // Always hide on theme change.  
    Visibility(Visibility::Collapsed);  
    //          ^^^^^^^^^^^ no ambiguity here  
}
```

There is no ambiguity in this case because of a special rule for unqualified name lookup that excludes functions, variables, and enumeration fields if the name is followed by a ::.

The case of a member with the same name as a type is called the Color Color problem, and the C# language “was specifically designed to permit” it. Unfortunately, the C++ language struggles with it.

Raymond Chen

Follow

