Resolving the ambiguity when your C++ class inherits from multiple base classes that have the same method

devblogs.microsoft.com/oldnewthing/20210813-05

August 13, 2021

Raymond Chen

Suppose you have a class which derives from two base classes, both of which have the same method.

```
struct Base1
{
    void Something();
    bool IsReady();
};

struct Base2
{
    void Something();
    bool IsReady();
};

struct Derived : Base1, Base2
{
};

void oops()
{
    Derived d;
    d.Something(); // ambiguous call to Something()
}
```

What happens next depends on what you want.

If you want the derived class to call both base class methods, then you get to write that out, specifying the order in which you want the calls to occur and how you want the results to be combined:

```
struct Derived : Base1, Base2
{
    void Something() { Base1::Something(); Base2::Something(); }
    bool IsReady() { return Base1::IsReady() && Base2::IsReady(); }
};
```

On the other hand, maybe you want everything to go to `Base1`:

```
struct Derived : Base1, Base2
{
    void Something() { Base1::Something(); }
    bool IsReady() { return Base1::IsReady(); }
};
```

Writing out this forwarder can get cumbersome if you have lots of methods, or if the methods have parameters that you need to perfect-forward:

```
struct Derived : Base1, Base2
{
    template<typename... Args>
    void Something(Args&&... args)
    { Base1::Something(std::forward<Args>(args)...); }

    template<typename... Args>
    bool IsReady(Args&&... args)
    { return Base1::IsReady(std::forward<Args>(args)...); }
};
```

Fortunately, there's a trick: You can steer the compiler toward the implementation in one of the base classes by using a `using` declaration.

```
struct Derived : Base1, Base2
{
    using Base1::Something;
    using Base1::IsReady;
};
```

One use of the `using` declaration is to import a method from a base class so you can add overloads to it. (If you don't import the methods from the base class, then your overloads *shadow* the base class methods.) Here, we are importing a method from a base class and not adding any overloads. We just want the method to be treated as if it were directly in our class all along.

This trick comes in handy in C++/WinRT stateful factories: The standard mechanism for implementing a stateful factory is to override the stateless factory methods provided by the `WidgetT` template with your own stateful ones.

```
namespace winrt::factory_implementation::Widget
{
    struct Widget : WidgetT<Widget, implementation::Widget, static_lifetime>
    {
        void Method(); // stateful static
    };
}
```

Windows Runtime static methods can be an aggregation of multiple independent providers that march together under one flag. Factoring out the functionality into separate classes is helpful if the class has a large number of static methods which fall naturally into categories of related methods. For example, there might be some static Widget members that use Plug and Play (a static `WidgetsChanged` event and a static `FindAllWidgets` method), and another group of static Widget methods that deal with system configuration (`SetMaximumActiveTime`, `AllowDownloadedContent`).

The naïve version doesn't work:

```
namespace winrt::factory_implementation::Widget
{
    struct WidgetPlugAndPlay
    {
        event_token WidgetsChanged(TypedEventHandler<IInspectable, IInspectable>
handler);
        void WidgetsChanged(event_token token);
        IVectorView<Widget> FindAllWidgets();
    };

    struct WidgetConfiguration
    {
        void SetMaximumActiveTime(TimeSpan limit);
        bool AllowDownloadedContent();
        void AllowDownloadedContent(bool value);
    };

    struct Widget : WidgetT<Widget, implementation::Widget, static_lifetime>,
        WidgetPlugAndPlay, WidgetConfiguration
    {
    };
}
```

It doesn't work because the static methods are multiply-inherited: Once from the `WidgetT` template (which forwards to static methods on the implementation class) and once from the `WidgetPlugAndPlay` and `WidgetConfiguration` base classes. To resolve the ambiguity, you can steer the methods with a `using` declaration.

```
namespace winrt::factory_implementation::Widget
{
    struct Widget : WidgetT<Widget, implementation::Widget, static_lifetime>,
        WidgetPlugAndPlay, WidgetConfiguration
    {
        using WidgetPlugAndPlay::WidgetsChanged;
        using WidgetPlugAndPlay::FindAllWidgets;

        using WidgetConfiguration::SetMaximumActiveTime;
        using WidgetConfiguration::AllowDownloadedContent;
    };
}
```

Related: How can I write a C++ class that iterates over its base classes?

Raymond Chen

Follow