# C++/WinRT implementation inheritance: Notes on winrt::implements, part 8

devblogs.microsoft.com/oldnewthing/20250228-00

February 28, 2025


We wrap up this series with a comparison of pros and cons.

Deriving your class from `winrt::implements` means that your declared interfaces will be implemented by anybody who derives from you. This is nice because it removes another point of failure: Forgetting to declare all the interfaces. (For example, if you add a new interface to your class, your derived classes don't all have to update.)

On the other hand, declaring the interface in `winrt::implements` means that your class must implement the entire interface, even if you wanted to delegate some of the methods to the derived class.

You can work around this with a virtual method or by using CRTP. Virtual methods are useful if you can dictate the method signature. CRTP gives the derived class more flexibility in deciding how to implement the method. However, you need to be careful with CRTP to avoid template code explosion.

If you instead choose a traditional C++ base class, then you can implement as much or as little of an interface as you like, and let the derived class implement the rest. The derived class does have to remember to declare the interface in its own `winrt::implements`, and the same remarks apply as above. Similarly, the same remarks regarding virtual methods and CRTP apply if you want your class to ask the derived class for help.

One case where CRTP is probably the best choice is if you need to call methods on a sibling interface implemented by the derived class.

```
template<typename D>
struct CanHideOnFocusLoss
{
    void HideOnFocusLoss(bool hide = true)
    {
        if (m_hide != hide) {
            m_hide = hide;
            if (m_hide) {
                // Call method on derived class
                m_lostFocusRevoker = owner()->LostFocus(
                    { this, &CanHideOnFocusLoss::OnLostFocus });
            } else {
                m_lostFocusRevoker.reset();
            }
        }
    }

private:
    winrt::LostFocus_revoker m_lostFocusRevoker;
    bool m_hide = false;

    D* owner() { return static_cast<D*>(this); }

    void OnLostFocus(winrt::IInspectable const&,
                     winrt::RoutedEventHandler const&)
    {
        // Call method on derived class
        owner()->Hide();
    }
};
```

The intention is that the implementation of an object that implements a `Flyout` can derive from `CanHideOnFocusLoss<T>` and enable auto-hide on focus loss.

```
// MyProject.idl

namespace Contoso
{
    runtimeclass MagicFlyout : Windows.UI.Xaml.Controls.Flyout
    {
        MagicFlyout();
    }
};

// MyProject.h

struct MagicFlyout :
    MagicFlyoutT<MagicFlyout>,
    CanHideOnFocusLoss<MagicFlyout>
{
    MagicFlyout()
    {
        HideOnFocusLoss(true);
    }
};
```

This is sort of a mixin-style base class that lets you attach behaviors to other classes.

The “deducing this” feature makes this more ergonomic for consumers, though it’s more cumbersome for the class itself.

```
struct CanHideOnFocusLoss
{
    template<typename D>
    void HideOnFocusLoss(this D&& self, bool hide = true)
    {
        if (m_hide != hide) {
            m_hide = hide;
            if (m_hide) {
                // Call method on derived class
                m_lostFocusRevoker = self.LostFocus(
                    { this, &CanHideOnFocusLoss::OnLostFocus<D> });
            } else {
                m_lostFocusRevoker.reset();
            }
        }
    }

private:
    winrt::LostFocus_revoker m_lostFocusRevoker;
    bool m_hide = false;

    template<typename D>
    void OnLostFocus(winrt::IInspectable const&,
                     winrt::RoutedEventHandler const&)
    {
        // Call method on derived class
        static_cast<D&&>(*this).Hide();
    }
};

// MyProject.h

struct MagicFlyout :
    MagicFlyoutT<MagicFlyout>,
    CanHideOnFocusLoss
{
    MagicFlyout()
    {
        HideOnFocusLoss(true);
    }
};
```

This ends a rather tedious discussion of class derivation patterns in C++/WinRT implementation classes, how you can `winrt::implements`, and why you might not want to. Most of it is fairly obvious once you get past the C++/WinRT template metaprogramming, but I felt compelled to write it all out for reference.