Using CRTP to your advantage: Simplifying overloaded Windows Runtime method projections in C++/WinRT

devblogs.microsoft.com/oldnewthing/20211208-00

December 8, 2021

Raymond Chen

C++/WinRT uses the so-called the curiously recurring template pattern, commonly known as CRTP. One of the nice features of CRTP is that the derived class method signature does not have to be a perfect match for the signature expected by the base class. All that matters is that the base class can call it *as if* it had the expected signature.[1]

It is common in the Windows Runtime to have overloads of the same method, where the extra parameters, if omitted, take on fixed default values:

```
namespace Contoso
{
    enum WidgetToggleOptions
    {
        Default,
        UseBothHands,
    };

    runtimeclass Widget
    {
        bool Toggle(); // defaults to WidgetToggleOptions.Default
        bool Toggle(WidgetToggleOptions options);
    }
}
```

The naïve implementation would go like this:

```
namespace winrt::Contoso::implementation
{
    struct Widget : WidgetT<Widget>
    {
        bool Toggle();
        bool Toggle(WidgetToggleOptions options);
    };

    bool Widget::Toggle()
    {
        return Toggle(WidgetToggleOptions::Default);
    }

    bool Widget::Toggle(WidgetToggleOptions options)
    {
        ... implementation ...
    }
}
```

For each overload, we implement a corresponding function in the CRTP derived class for the base class to forward to.

But you're working too hard.

The base class is going to call `Widget::Toggle()`, but that doesn't mean that the signature must be exactly `Widget::Toggle()`. You have been taking advantage of this already: A method whose Windows Runtime signature is `void Method(String name)` can be implemented with any of these signatures:

```
void Method(winrt::hstring name);
void Method(winrt::hstring const& name);
winrt::fire_and_forget Method(winrt::hstring name);
```

So let's take advantage of it some more: We can write one implementation that covers both projected methods by using default parameters.

```
namespace winrt::Contoso::implementation
{
    struct Widget : WidgetT<Widget>
    {
        bool Toggle(WidgetToggleOptions options = WidgetToggleOptions::Default);
    };

    bool Widget::Toggle(WidgetToggleOptions options)
    {
        ... implementation ...
    }
}
```

When the CRTP base class tries to call `Widget::Toggle()`, it will call the `Toggle(WidgetToggleOptions options)` method, using the default parameter of `WidgetToggleOptions::Default` to fill in the missing explicit parameter.

I think this approach is easier to read, especially when there are multiple overloads with longer and longer parameter lists, since it lets you see the behavior of the short-parameter-list versions at a glance, and avoids any risk of the various overloads falling out of sync.

[1] This is similar to the rule in the C++ language specification (**[namespace.std]**) which permits standard library functions to have additional default function parameters or default template parameters, as long as they are callable via the standard-prescribed signatures. (There is an exception for so-called *addressable functions*, which must have exactly the standard-specified signature.) This rule exists so that the implementation can use these default parameters to control which functions participate in overloading, or which templates can be instantiated (via SFINAE). The way the rule is expressed in the standard is “A program may not take the address of a standard library function.”

Raymond Chen

Follow