# Exploiting C++/WinRT CRTP: Property and event declarations

devblogs.microsoft.com/oldnewthing/20230317-00

March 17, 2023

Raymond Chen

In C++/WinRT, properties are represented by method calls. Given this Windows Runtime class definition:

```
namespace MyNamespace
{
    runtimeclass Widget
    {
        Double Height;
    }
}
```

you can access the property like so:

```
Widget w;

// 0 parameters = get value
auto height = w.Height();

// 1 parameter = set value
w.Height(newHeight);
```

If you implement this class in C++/WinRT, the auto-generated header files use the Curiously Recurring Template Pattern (commonly known as CRTP) and expect your implementation class to follow the same pattern:

```
namespace winrt::MyNamespace::implementation
{
    struct Widget : WidgetT<Widget>
    {
        double m_height;
        auto Height() const { return m_height; }
        void Height(double const& height) { m_height = height; }
    };
}
```

If you have a lot of properties, writing these accessor methods can be quite tedious and therefore error-prone.

But we can exploit CRTP to make it easier.

You see, the CRTP wrappers don't actually require `Height()` to be a method. It just requires that `Height()` be an expression that represents the property value. Similarly, `Height(newValue)` just needs to be an expression whose side effect is changing the property value.

```
// This code is autogenerated by C++/WinRT

int32_t get_Height(double* value) noexcept final try
{
    typename D::abi_guard guard(this->shim());
    *value = detach_from<double>(this->shim().Height());
    return 0;
}
catch (...) { return to_hresult(); }

int32_t put_Height(double value) noexcept final try
{
    typename D::abi_guard guard(this->shim());
    this->shim().Height(value);
    return 0;
}
catch (...) { return to_hresult(); }
```

In words, the autogenerated code expects `d.Height()` to produce something that can be stored in a `double`, and it expects `d.Height(value)` to do something with the `value`.

These look like method calls, but they don't have to be. They can be anything, provided that sequence of tokens produces a valid result.

Specifically, they can be callable objects.

```
template<typename T>
struct winrt_property
{
    winrt_property(T const& initial =
        winrt_empty_value<T>()) :
        value(initial) {}

    T operator()() const { return value; }
    void operator()(T const& newValue) { value = newValue; }

    T value;
};

namespace winrt::MyNamespace::implementation
{
    struct Widget : WidgetT<Widget>
    {
        winrt_property<double> Height;
    };
}
```

When the C++/WinRT library does `this->shim().Height()`, this obtains the `Widget` implementation class and then looks up `Height`, which is a member variable, and then applies the `()` function call operator, which we overloaded to return the current wrapped value.

Similarly, when the C++/WinRT library does `this->shim().Height(value)`, this invokes the overloaded single-parameter function call operator which updates the value.

In the implementation, you can access the value either by using the function call operator:

```
void Widget::IncreaseHeight(double increment)
{
    auto currentHeight = Height();
    Height(currentHeight + increment);
}
```

or you can access the `value` member, which was intentionally left public for this purpose.

```
void Widget::IncreaseHeight(double increment)
{
    Height.value += increment;
}
```

You can use the same trick for C++/WinRT event implementations.

```
template<typename D>
struct winrt_event : winrt::event<D>
{
    winrt_event() = default;

    using winrt::event<D>::operator();
    auto operator()(D const& handler)
        { return this->add(handler); }
    void operator()(winrt::event_token const& token)
        { this->remove(token); }
};

// Class definition

namespace MyNamespace
{
    runtimeclass Widget
    {
        Windows.Foundation.TypedEventHandler<Widget, Object> Closed;
    }
}

// Implementation
namespace winrt::MyNamespace::implementation
{
    // Just an abbreviation to save some typing.
    using ClosedHandler = Windows::Foundation::TypedEventHandler<
        MyNamespace::Widget, Windows::Foundation::IInspectable>;

    struct Widget : WidgetT<Widget>
    {
        winrt_event<ClosedHandler> Closed;
    };
}
```

Note the need to namespace-qualify `Widget` when it appears as the first template type parameter of `TypedEventHandler` because we want it to be the projected type and not the implementation type.