How can I call a method on a derived class from a base class, say, to get a strong reference to the containing object?

devblogs.microsoft.com/oldnewthing/20230120-21

January 20, 2023

Raymond Chen

Suppose you have a base class, and you want to get a strong reference to your derived class. You may need to do this, for example, if your base class has a method which is a coroutine, and it needs to retain a strong reference to itself so that it can extend its lifetime into the coroutine body, thereby preventing the object from being destroyed after the coroutine reaches its first suspension point.

```
// C++/WinRT style

struct Base
{
    winrt::IAsyncAction DoSomethingAsync()
    {
        auto lifetime = get_strong(); // ????

        co_await this->step1();

        this->step2();
    }
};

struct Derived : DerivedT<Derived>, Base
{
};

// PPL style

struct Base
{
    Concurrency::task<void> DoSomethingAsync()
    {
        auto lifetime = shared_from_this(); // ????

        co_await this->step1();

        this->step2();
    }
};

struct Derived : std::enable_shared_from_this<Derived>, Base
{
};
```

First, let's forget about coroutines. They are the motivation for the question, but they aren't really relevant to the solution.

If you have an instance of a base class and you want to get an instance of the derived class, what can you do?

Well, if you know what the derived class is, you can downcast to it.

```
// C++/WinRT style
struct Base
{
    void BaseMethod()
    {
        auto lifetime = static_cast<Derived*>(this)->get_strong();

        ...
    }
};

// PPL style

struct Base
{
    void BaseMethod()
    {
        auto lifetime = static_cast<Derived*>(this)->shared_from_this();

        ...
    }
};
```

This trick works only if you are absolutely certain that the `Base` is the base portion of a `Derived`. Now, maybe you want to use the `Base` as the base of multiple derived classes. In that case, you can pass the derived class as a template parameter, turning this into a case of CRTP:

```
// C++/WinRT style

template<typename D>
struct Base
{
    void BaseMethod()
    {
        auto lifetime = static_cast<D*>(this)->get_strong();

        ...
    }
};

struct Derived : DerivedT<Derived>, Base<Derived>
{
};

// PPL style

template<typename D>
struct Base
{
    void BaseMethod()
    {
        auto lifetime = static_cast<D*>(this)->shared_from_this();

        ...
    }
};

struct Derived : std::enable_shared_from_this<Derived>, Base<Derived>
{
};
```

Another solution is to give the `Base` a weak reference to the container. This removes the need to know what the container is.

```
// C++/WinRT style

struct Base
{
    winrt::weak_ptr<IInspectable> derived_weak;

    void BaseMethod()
    {
        auto lifetime = derived_weak.get();

        ...
    }
};

struct Derived : DerivedT<Derived>, Base
{
    Derived()
    {
        derived_weak = get_weak();
    }
};

// PPL style

template<typename D>
struct Base
{
    std::weak_ptr<void> derived_weak;

    void BaseMethod()
    {
        auto lifetime = derived_weak.lock();

        ...
    }
};

struct Derived : std::enable_shared_from_this<Derived>, Base
{
    Derived()
    {
        // doesn't work!
    }
};
```

Uh-oh, we're kind of stuck in the PPL case because the weak pointer hiding inside `enable_shared_from_this` is not initialized at the point the `Derived` is constructed. (It gets set by `std::make_shared` after the object has been constructed.)

Another option is to have a pure virtual method which derived classes must implement in order to provide the necessary strong pointer.

```
// C++/WinRT style

struct Base
{
    virtual winrt::IInspectable derived_strong() const = 0;

    void BaseMethod()
    {
        auto lifetime = derived_strong();

        ...
    }
};

struct Derived : DerivedT<Derived>, Base
{
    winrt::IInspectable derived_strong() const override
    {
        return *this;
    }
};

// PPL style

struct Base
{
    virtual std::shared_ptr<const void> derived_strong() const = 0;

    void BaseMethod()
    {
        auto lifetime = derived_strong();

        ...
    }
};

struct Derived : std::enable_shared_from_this<Derived>, Base
{
    std::shared_ptr<const void> derived_strong() const override
    {
        return shared_from_this();
    }
};
```

And C++23's "deducing this" adds another option:

```
// C++/WinRT style

struct Base
{
    template<typename Derived>
    winrt::IAsyncAction DoSomethingAsync(this Derived&& self)
    {
        auto lifetime = self.get_strong();

        co_await this->step1();

        this->step2();
    }
}

struct Derived : DerivedT<Derived>, Base
{
};

// PPL style

struct Base
{
    template<typename Derived>
    Concurrency::task<void> DoSomethingAsync(this Derived&& self)
    {
        auto lifetime = self.shared_from_this();

        co_await this->step1();

        this->step2();
    }
};

struct Derived : std::enable_shared_from_this<Derived>, Base
{
};
```

Which should you choose?

Well, it's up to you. My preference is to use the CRTP pattern for C++/WinRT types, since that avoids extra data members and vtables, and it is consistent with other C++/WinRT patterns. And that leads me to prefer the CRTP pattern for the PPL case, too, for consistency.

And if you can assume C++23, then the "deducing this" form lets you get the benefits of CRTP without having to do CRTP. So that's my first choice.

But that's just my opinion. You might prefer something else.

Bonus chatter: You might think of having `Base` derive from `enable_shared_from_this`. That would work, but it also means that `Derived` cannot derive from `enable_shared_from_this`, and it also means that `Derived` cannot derive from both `Base1` and `Base2` if both of the base classes derive from `enable_shared_from_this`. The general convention is that the most derived class is the one that gets to derive from `enable_shared_from_this`.