Inside STL: The shared_ptr constructor and enable_shared_from_this

devblogs.microsoft.com/oldnewthing/20230816-00

August 16, 2023

Raymond Chen

If you create a class of the form

```
struct S : std::enable_shared_from_this<S>
{
    /* ... */
};
```

which derives from `std::enable_shared_from_this` of itself (using the curiously recurring template pattern), then this class becomes a candidate for special treatment by `shared_ptr`: The `shared_from_this()` method will produce a `shared_ptr<S>`. Some restrictions apply.

Here's how it works.

```
template<typename T>
struct enable_shared_from_this
{
    using esft_detector = enable_shared_from_this;
    std::weak_ptr<T> weak_this;

    std::weak_ptr<T> weak_from_this()
    { return weak_this; }

    std::shared_ptr<T> shared_from_this()
    { return weak_this.lock(); }

};
```

When you derive from `enable_shared_from_this`, you get a secret weak pointer which the C++ standard calls `weak_this`. The inherited member function `weak_from_this()` returns that weak pointer, and the inherited member function `strong_from_this()` returns a strong version of that weak pointer.

Who initializes this weak pointer?

When the control block is created, the `shared_ptr<S>` constructor snoops at the object that is being managed by the control block. If it uniquely inherits from `std::enable_shared_from_this` and does so publicly, then the constructor stashes a weak pointer to the newly-constructed `shared_ptr` in `weak_this`.

That's the only time it happens. If anything goes wrong, you don't get your `weak_this`, and the `weak_from_this()` and `shared_from_this()` methods throw a "bad weak reference" exception.

Here are some things that could go wrong:

- The `S` object was never created as part of a `shared_ptr`. Maybe it was created as a local variable or as a member of a larger structure.
- The `S` object derives from `std::enable_shared_from_this`, but the base class was not *public*.
- The `S` object derives from `std::enable_shared_from_this` *more than once*.

Some time ago, I discussed a way to make sure people use `make_shared` to make the object, which you can use to reduce the likelihood of the first problem.

The second problem is often an oversight, forgetting that base classes of a `class` are private by default. (Base classes of a `struct` are public by default.)

The third problem is a more complex oversight which usually comes about when you build a derivation hierarchy out of multiple pieces, unaware that some of the pieces are already using `std::enable_shared_from_this`.

Okay, so that's what it does, but how does it work?

The `shared_ptr` constructor detects the presence of a unique `std::enable_shared_from_this` base class by using the `esft_detector` that I put in the expository declaration.

```
template<typename T, typename = void>
struct supports_esft : std::false_type {};

template<typename T>
struct inline bool supports_esft<T,
    std::void_t<typename T::esft_detector>>
    : std::true_type {};
```

Our first attempt at detecting `std::enable_shared_from_this` support is checking whether our marker type `esft_detector` is available. If there is no `std::enable_shared_from_this` in the derivation hierarchy, then the type will be missing outright. If it is present but not `public`, then the check will fail due to the type being inaccessible.

The code that sets the weak pointer uses this detector helper:

```
template<typename T, typename D>
struct shared_ptr
{
    shared_ptr(T* ptr)
    {
        ... do the usual stuff ...

        /* Here comes enable_shared_from_this magic */
        if constexpr (supports_esft<T>::value) {
            using detector = T::esft_detector;
            ptr->detector::weak_this = *this;
        }
    }

    ... other constructors and stuff ...
};
```

If the `esft_detector` is present, then we use it to tell us which specialization of `std::enable_shared_from_this` was used, so that we can set that base class's `weak_this`.

We can't stop here, though, because this results in a compilation error if there are multiple `std::enable_shared_from_this` base classes.

```
struct B : std::enable_shared_from_this<B> {};
struct M1 : B {};
struct M2 : B {};
struct D : M1, M2 {};

auto p = std::make_shared<D>();

error: ambiguous reference to base class at

ptr->detector::weak_this = *this;
     ^^^^^^^^
```

To avoid this, we also ensure that the detector is *unique*.

```
template<typename T>
struct inline bool supports_esft<T,
    std::void_t<typename T::esft_detector>>
    : std::is_convertible<T *, typename T::esft_detector *>::type {};
```

If a pointer to `T` is convertible to a pointer to the detector, then we know that the detector appears only once among the base classes of `T`.

One could argue that instead of silently ignoring the cases where `std::enable_shared_from_this` was declared but could not be used, the language could have said that such a program is ill-formed and produces a compiler error. But no, the language says that if you break rules 2 or 3, then the `std::enable_shared_from_this` is simply ignored, and you are left scratching your head trying to figure out where you went astray.

I suspect part of the problem is that it is explicitly legal to use `shared_ptr<T>` when `T` is an incomplete type.