On detecting improper use of std::enable_shared_from_this

devblogs.microsoft.com/oldnewthing/20231013-00

October 13, 2023

Raymond Chen

We saw some time ago that you must publicly derive from `std::enable_shared_from_this` in order for `shared_from_this()` to work. Can we fix it so that the problem is detected at compile time rather than failing mysteriously at runtime?

```
template<class T>
class enable_shared_from_this
{
    static_assert(
        std::is_convertible_v<T*,
                              enable_shared_from_this*>,
        "You must publicly derive from "
        "enable_shared_from_this exactly once");

public:
    using esft_tag = enable_shared_from_this;

    [[nodiscard]] shared_ptr<T> shared_from_this() {
        return shared_ptr<T>(weak);
    }

    [[nodiscard]] shared_ptr<T> shared_from_this() const {
        return shared_ptr<const T>(weak);
    }

    [[nodiscard]] weak_ptr<T> weak_from_this() noexcept {
        return weak;
    }

    [[nodiscard]] weak_ptr<const T> weak_from_this() const noexcept {
        return weak;
    }

protected:
    constexpr enable_shared_from_this() noexcept : weak() {}

    enable_shared_from_this(const enable_shared_from_this&) noexcept {}
    enable_shared_from_this& operator=(const enable_shared_from_this&) noexcept
    { return *this; }

private:
    template<typename U> friend class shared_ptr;

    mutable weak_ptr<T> weak;
};
```

We assert that the class `T` publicly and uniquely derives from `enable_shared_from_this<T>`. Unfortunately, this doesn't work because we are asserting too soon:

```
class something : std::enable_shared_from_this<something>
//   instantiated ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
{
    ⟦ ... ⟧
};
```

At the point that `enable_shared_from_this<someting>` is instantiated, the class `something` is incomplete.

We can defer the assertion to the constructor, because the constructor body is instantiated after the completion of the derived class.

```
template<class T>
class enable_shared_from_this
{
public:
    using esft_tag = enable_shared_from_this;

    [[nodiscard]] shared_ptr<T> shared_from_this() {
        return shared_ptr<T>(weak);
    }

    [[nodiscard]] shared_ptr<T> shared_from_this() const {
        return shared_ptr<const T>(weak);
    }

    [[nodiscard]] weak_ptr<T> weak_from_this() noexcept {
        return weak;
    }

    [[nodiscard]] weak_ptr<const T> weak_from_this() const noexcept {
        return weak;
    }

protected:
    constexpr enable_shared_from_this() noexcept : weak() {
      static_assert(
          std::is_convertible_v<T*,
                                enable_shared_from_this*>,
          "You must publicly derive from "
          "enable_shared_from_this exactly once");

    }

    enable_shared_from_this(const enable_shared_from_this&) noexcept {}
    enable_shared_from_this& operator=(const enable_shared_from_this&) noexcept
    { return *this; }

private:
    template<typename U> friend class shared_ptr;

    mutable weak_ptr<T> weak;
};
```

But wait, there's this other mistake: Deriving from two different specializations of `enable_shared_from_this`:

```
struct B1 : std::enable_shared_from_this<B1> {};
struct B2 : std::enable_shared_from_this<B2> {};

struct D : B1, B2 {};

// No complaint!
auto p = std::make_shared<D>();

// This throws bad_weak_ptr
auto b1 = p->B1::shared_from_this();
```

The `make_shared<D>()` executes just fine, but the resulting object's `std::enable_shared_from_this<B1>` and `std::enable_shared_from_this<B2>` objects are nonfunctional because `D` derived multiple times from `std::enable_shared_from_this`. We failed to detect this in our `static_assert` because the `static_assert` checks only that each individual specialization is unique.

Remember, `enable_shared_from_this` is not referring to "all specializations of this template". It is referring to the specific specialization being instantiated, since it is the injected class name. In other words, the full version of the static assertion is

```
static_assert(
    std::is_convertible_v<T*,
                          enable_shared_from_this<T>*>,
    "You must publicly derive from "
    "enable_shared_from_this exactly once");
```

But we want to check *all* specializations, not just the one being instantiated. How can we do that?

We can take advantage of the `esft_tag` that is used by `make_shared()` to locate the `enable_shared_from_this()` base class.

```
static_assert(
    std::is_convertible_v<T*,
                          typename T::esft_tag*>,
    "You must publicly derive from "
    "enable_shared_from_this exactly once");
```

If `T` has multiple base classes which are specializations of `enable_shared_from_this<T>`, there are two cases:

- All of the base classes are the same `enable_shared_from_this<T>`. In this case, the conversion from `T*` to `enable_shared_from_this<T>*` is ambiguous, and `is_convertible_v` returns `false`.

- There are two base classes which are different specializations, say `enable_shared_from_this<B1>` and `enable_shared_from_this<B2>` (where `B1` ≠ `B2`). In this case, tthe nested type `T::esft_tag` will have conflicting definitions, and you get an ambiguous type error.

The standard `enable_shared_from_this<T>` doesn't do this extra enforcement because you are allowed to specialize `enable_shared_from_this<T>` with an incomplete type!

```
struct D;

struct B : std::enable_shared_from_this<D>
{
};

struct D : B
{
};

auto p = std::make_shared<D>();
auto q = p->shared_from_this();
```

We made `B` derive from `std::enable_shared_from_this<D>` even though we don't know what `D` is yet. This is legal. It just means that if you ever decide to put `B` inside a `shared_ptr`, it had better be a base class of a larger `D` object.

So let's just declare that case as out of scope for our "strict `enable_shared_from_this`".

Can we augment the standard `enable_shared_from_this` to add this sort of safety checking? Let's try it:

```
template<typename T>
struct strict_enable_shared_from_this
    : std::enable_shared_from_this<T>
{
    using esft_tag = std::enable_shared_from_this<T>;
};

template<typename T, typename... Args>
std::shared_ptr<T>
strict_make_shared(Args&&... args)
{
    static_assert(
        std::is_convertible_v<T*,
                              typename T::esft_tag*>,
        "You must publicly derive from "
        "strict_enable_shared_from_this exactly once");
    return std::make_shared<T>(std::forward<Args>(args)...);
}
```

If we can be sure that everybody uses `strict_enable_shared_from_this`, then `strict_make_shared` can verify that the resulting `shared_ptr` owns an object which indeed holds a weak pointer to itself. On the other hand, if somebody sneaks in and uses a standard `enable_shared_from_this`, then they can avoid the detection.

```
struct B1 : strict_enable_shared_from_this<D> {};
struct B2 : std::enable_shared_from_this<D> {};
struct D: B1, B2 {};

// No complaint, but the shared_from_this method fails.
auto p = strict_make_shared<D>();
```

It would be nice if the C++ standard library had a type trait

```
template<typename T>
struct can_enable_shared_from_this;

template<typename T>
inline constexpr bool can_enable_shared_from_this_v =
    can_enable_shared_from_this<T>::value;
```

Then we can we can define our own (or maybe the C++ standard libray can also provide)

```
template<typename T, typename... Args>
std::shared_ptr<T>
make_shared_with_weak_ptr(Args&&... args)
{
    static_assert(
        std::can_enable_shared_from_this_v<T>);
    return std::make_shared<T>(std::forward<Args>(args)...);
}
```