What it means when you convert between different shared_ptrs

devblogs.microsoft.com/oldnewthing/20230817-00

August 17, 2023

Raymond Chen

The C++ `shared_ptr` manages a reference-counted pointer. Usually, it's a pointer to an object that will be `delete`'d when the last reference expires. But it doesn't have to be that way.

Recall that a `shared_ptr` is really two pointers.

- A pointer to a *control block* which manages the shared and weak reference counts, and which destroys an object (commonly known as the *managed object*) when the shared reference count drops to zero.
- A pointer to return from the `get()` method, commonly know as the *stored pointer*.

Most of the time, the stored pointer points to the managed object, because that what you get when you construct a `shared_ptr`s from a raw pointer or when you call `make_shared`. But what is the use case for a `shared_ptr` where the managed object and the stored pointer are different?

You may want to have a `shared_ptr` whose `get()` returns a pointer to a sub-object of another large object. In that case, the managed object is the larger object, and the stored pointer is to the sub-object.

```
struct Sample
{
    int value1;
    int value2;
};

void consume(std::shared_ptr<int> pint);

std::shared_ptr<Sample> p = std::make_shared<Sample>();
consume(std::shared_ptr<int>(p, &p->value1));

// Or, more tersely
auto p = std::make_shared<Sample>();
consume({ p, &p->value1 });
```

In the above example, we have a class `Sample` with two members. We create a `shared_ptr` to that class and save it in `p`. But say there's another function that wants a `shared_ptr<int>`. No problem, we can convert the `std::shared_ptr<Sample>` into a `std::shared_ptr<int>` by reusing the control block (first parameter `p`) and substituting a new stored pointer (second parameter `&p->value1`). The `consume` function can use the `shared_ptr<int>` to access the `value1` member, and the control block of that `shared_ptr<int>` prevents the `Sample` from being destroyed, which in turn prevents the `value1` from being destroyed.

The general principle is that the lifetime of the stored pointer should be contained with the lifetime of the managed object. It can be a direct containment relationship, like we did with `value1`, or it could be a more complex chain of lifetime dependencies.

```
struct Other
{
    int value;
};

struct Sample2
{
    const std::unique_ptr<Other> m_other =
        std::make_unique<Other>();
};

auto p = std::make_shared<Sample2>();
consume({ p, &p->m_other->value });
```

In this second example, the stored pointer of the `shared_ptr<int>` we pass to the `consume()` function points to the `value` member inside the `Other` object to which the `Sample2` holds a unique pointer. The control block in that `shared_ptr<int>` controls the lifetime of the `Sample2` object, which is acceptable because as long as the `Sample2` object remains alive, the `value` inside the `Other` will be alive.

Now, the compiler doesn't check that you have a positive chain of lifetime control from the managed object to the stored pointer. You could do something silly like

```
struct Sample3
{
    std::unique_ptr<Other> m_other =
        std::make_unique<Other>();
};

auto p = std::make_shared<Sample3>();
consume({ p, &p->m_other->value });
p->m_other = nullptr; // oops, chain is broken
```

and the `shared_ptr<int>` will think it's keeping the `value` alive, even though you broke the link from the `Sample3` to the `Other`.

Or you can do something even sillier like

```
int unrelated;
consume({ p, &unrelated });
```

and the `shared_ptr<int>` will access `unrelated` even though its lifetime is unrelated to the `Sample2`. If `unrelated` is destroyed, the `shared_ptr<int>` will have a dangling stored pointer.

These `shared_ptr` objects in which the managed object is different from the pointed-to object are commonly known as *aliasing* shared pointers.

Okay, so I showed one way of creating an aliasing shared pointer, namely by constructing a `shared_ptr` from an existing `shared_ptr` (which shares the managed object) and providing a different stored pointer. If the new stored pointer points to a base class of the original, then the `shared_ptr` has a conversion operator that creates an aliasing shared pointer to the base-class subobject.

```
struct Base
{
};

struct Derived : Base
{
};

std::shared_ptr<Derived> p = std::make_shared<Derived>();
std::shared_ptr<Base> b = p; // auto-conversion
// equivalent to
std::shared_ptr<Base> b(p, p.get());
```

If you want to do the reverse conversion (from `Base` to `Derived`), you can write it out explicitly:

```
std::shared_ptr<Derived> b(p, static_cast<Derived*>(p.get()));
```

Of course, this requires that the stored `Base` pointer really is a pointer to the `Base` part of a larger `Derived` object.

The C++ language comes with some helper functions that construct a `shared_ptr` by casting the stored pointer of another `shared_ptr`.

Helper	**Equivalent to `std::shared_ptr<T>{...}`**
`std::static_pointer_cast<T>(p)`	`{ p, static_cast<T*>(p.get()) }`
`std::const_pointer_cast<T>(p)`	`{ p, const_cast<T*>(p.get()) }`

`std::reinterpret_pointer_cast<T>(p)`	`{ p, reinterpret_cast<T*>(p.get()) }`
`std::dynamic_pointer_cast<T>(p)`	`{ p, dynamic_cast<T*>(p.get()) }`

Everything looks great until we get to `dynamic_pointer_cast`, which is *not* equivalent to a one-liner that uses `dynamic_cast`!

The reason is that, unlike the other casts, `dynamic_cast` can change a non-null pointer to a null pointer, which happens if the runtime type does not match. In that case, the `dynamic_pointer_case` returns an empty shared pointer (rather than a shared pointer with a control block and no stored pointer), because there is nothing whose lifetime needs to be extended.

Now we can finish that table:

Helper	**Equivalent to `std::shared_ptr<T>{...}`**
`std::static_pointer_cast<T>(p)`	`{ p, static_cast<T*>(p.get()) }`
`std::const_pointer_cast<T>(p)`	`{ p, const_cast<T*>(p.get()) }`
`std::reinterpret_pointer_cast<T>(p)`	`{ p, reinterpret_cast<T*>(p.get()) }`
`std::dynamic_pointer_cast<T>(p)`	`{ p, dynamic_cast<T*>(p.get()) }` (if cast succeeds)
	`{}` (if cast fails)

This wrinkle about control blocks for null pointers does call out that two boxes in the shared pointer diagram are technically legal though strange.

	Null control block	**Non-null control block**
Null stored pointer	Empty	Phantom (?)
Non-null stored pointer	Indulgent (?)	Full

So far, we've been dealing with empty shared pointers (that manage no object and have no stored pointer) and full shared pointers (that manage an object and have a stored pointer). But there are two other boxes, which I've named "Phantom" and "Indulgent". We'll look at those two weird guys next time.