Inside STL: Smart pointers

devblogs.microsoft.com/oldnewthing/20230814-00

August 14, 2023

Raymond Chen

The C++ standard library comes with a few smart pointer types.

The simplest one is `unique_ptr`: This class babysits a raw pointer and remembers to delete it at destruction (in an appropriate manner). Dumping the contents of a `unique_ptr` is just looking at the raw pointer inside.

The complication is that there is also a deleter object in the `unique_ptr`. This deleter object is usually an empty class, so it is stored as part of a compressed pair.

The Visual Studio debugger has a visualizer that understands `unique_ptr`, but here's what it looks like at a low level in the Microsoft implementation:

```
0:000< ?? p
class std::unique_ptr<S,std::default_delete<S> >
   +0x000 _Mypair          : std::_Compressed_pair<std::default_delete<S>,S *,1>
0:000< ?? p._Mypair
class std::_Compressed_pair<std::default_delete<S>,S *,1>
   +0x000 _Myval2          : 0x0000020a`11f08490 S
0:000< ?? p._Mypair._Myval2
struct S * 0x0000020a`11f08490 ← here is the unique object
   +0x000 a                : 0n42
```

Next up is are the buddies `shared_ptr` and `weak_ptr`. These guys are just alternate policies around the same design, which centers around a "control block".

```
struct control_block
{
    virtual void Dispose() = 0;
    virtual void Delete() = 0;
    std::atomic<unsigned long> shareds;
    std::atomic<unsigned long> weaks;
};

template<typename T>
struct shared_ptr
{
    T* object;
    control_block* control;
};

template<typename T>
struct weak_ptr
{
    T* object;
    control_block* control;
};
```

The control block keeps track of how many shared and weak pointers exist, but in a somewhat unusual way. The naïve approach would be to have `shareds` and `weaks` count the number of shared and weak pointers, respectively. However, this ends up being more complicated because detecting when the last pointer to the control block has been destroyed would require us to perform atomic checks on *two* member variables simultaneously.

You can solve this by having `weaks` be the number of shared pointers *plus* the number of weak pointers. This is the total number of things keeping the control block alive. A test of a single atomic variable will tell you when the last pointer has been destroyed.

Here's how the fundamental operations play out with this design:

```
struct control_block
{
    virtual void Dispose() = 0;
    virtual void Delete() = 0;
    std::atomic<unsigned long> shareds;
    std::atomic<unsigned long> refs;
};
```

Create the first shared pointer: Create a new control block with `shareds` and `refs` both initialized to 1.

Copy a nonempty weak pointer: Atomically increment the `refs`. (We know that the `refs` is at least one, because it includes the weak pointer we are copying from.)

Copy a nonempty shared pointer: Atomically increment both `shareds` and `refs`. (We know that both are already at least one, because they includes the shared pointer we are copying from.)

Convert a nonempty shared pointer to a weak pointer: Atomically increment the `refs`. (We know that the `refs` is at least one, because it includes the weak pointer we are copying from.)

Convert a nonempty weak pointer to a shared pointer: Atomically increment the `shareds`, but only if it was originally nonzero.[1] If successful, then also increment the `refs`. If `shareds` was zero, then produce an empty shared pointer instead.

Destruct a nonempty weak pointer: Atomically decrement the `refs`. If this decrements to zero, then call `Delete()` to delete the control block.

Destruct a nonempty shared pointer: Atomically decrement the `shareds`. If this decrements to zero, then call `Dispose()` to destruct the managed object. Next, atomically decrement the `refs`. If this also decrements to zero, then call `Delete()` to delete the control block.

This design works, but it turns out you can be more clever about it: Instead of counting all shared pointers toward the `refs`, count only one shared pointer toward the `refs`, and update `refs` only when the number of `shareds` transitions between 1 and 0.

So here's our optimized version:

```
struct control_block
{
    virtual void Dispose() = 0;
    virtual void Delete() = 0;
    std::atomic<unsigned long> shareds;
    std::atomic<unsigned long> weaks;
};
```

Create the first shared pointer: Create a new control block with `shareds` and `weaks` both initialized to 1.

Copy a nonempty weak pointer: Atomically increment the `weaks`. (We know that the `weaks` is at least one, because it includes the weak pointer we are copying from.)

Copy a nonempty shared pointer: Atomically increment the `shareds`. (We know that the `shareds` is at least one, because it includes the shared pointer we are copying from.)

Convert a nonempty shared pointer to a weak pointer: Atomically increment the `weaks`. (We know that the `weaks` is at least one, because it includes the weak pointer we are copying from.)

Convert a nonempty weak pointer to a shared pointer: Atomically increment the `shareds`, but only if it was originally nonzero.¹ If it was zero, then produce an empty shared pointer instead.

Destruct a nonempty weak pointer: Atomically decrement the `weaks`. If this decrements to zero, then call `Delete()` to delete the control block.

Destruct a nonempty shared pointer: Atomically decrement the `shareds`. If this decrements to zero, then there's extra work to do: Destroy the associated object and decrement the `weaks`. If `weaks` also decrements to zero, then call `Delete()` to delete the control block.

There's a lesser-known feature of shared and weak pointers: The *aliasing contructor*. This lets you create a `shared_ptr` that dereferences to an object that is different from the one managed by the control block. For example, you can do this:

```
struct S { int a = 42; int b = 99; };

auto s = std::make_shared<S>();
auto i = std::shared_ptr<int>(s, &s->b);
```

The shared pointer `i` points to the `b` member of the shared `S` object, but using the lifetime of the `S` object controlled by the pre-existing shared pointer `s`.

Internally, you create an aliasing shared pointer by using the same control block as the source shared pointer (incrementing the `shareds`, naturally), but setting the object pointer to the second constructor parameter.

Again, the Visual Studio debugger understands all of this and gives you a nice visualization. If you're debugging at a lower level, then here's what it looks like in the debugger:

```
0:000> ?? p2
class std::shared_ptr<S>
   +0x000 _Ptr             : 0x0000020a`11f084e0 S
   +0x008 _Rep             : 0x0000020a`11f084d0 std::_Ref_count_base
0:000> ?? p2._Ptr
struct S * 0x0000020a`11f084e0
   +0x000 a                : 0n42
0:000> ?? p2._Rep
class std::_Ref_count_base * 0x0000020a`11f084d0
   +0x000 __VFN_table : 0x00007ff7`9e788758
   +0x008 _Uses            : 1
   +0x00c _Weaks           : 1
```

Here's how the names we used map to the names in the Microsoft implementation of the C++ standard library:

Our name	MSVC name	Notes
`object`	`_Ptr`	Pointer to managed object
`control`	`_Rep`	Pointer to control block
`shareds`	`_Uses`	Number of shared pointers
`weaks`	`_Weaks`	Number of weak pointers + 1 if there are any shared pointers

When inspecting a `shared_ptr`, you can just follow the `_Ptr` to get to the managed object. You know that the pointer is valid because this is a shared pointer which keeps the object alive. (If the `_Ptr` is `nullptr`, then the shared pointer is empty.)

A weak pointer looks pretty much identical in the debugger:

```
0:000> ?? p3
class std::weak_ptr<S>
   +0x000 _Ptr             : 0x0000020a`11f084e0 S
   +0x008 _Rep             : 0x0000020a`11f084d0 std::_Ref_count_base
0:000> ?? p3._Ptr
struct S * 0x0000020a`11f084e0
   +0x000 a                : 0n42
0:000> ?? p3._Rep
class std::_Ref_count_base * 0x0000020a`11f084d0
   +0x000 __VFN_table : 0x00007ff7`9e788758
   +0x008 _Uses            : 1
   +0x00c _Weaks           : 2
```

When inspecting a weak pointer in the debugger, you have to check the control block's shared pointer count to know whether or not the object is still valid. If the shared pointer count is zero, then the weak pointer has expired, and the object pointer points to freed memory.

Next time, we'll look at `std::make_shared` and `std::enable_shared_from_this`.

Bonus viewing: Stephan T. Lavavej's lecture on `shared_ptr` and `unique_ptr`. He also gave an advanced lecture, which I cannot find.

¹ This can be accomplished by a `compare_exchange` loop. Given the scenario, this would be better served by a `compare_exchange_weak`.