Inside STL: The different types of shared pointer control blocks

devblogs.microsoft.com/oldnewthing/20230821-00

August 21, 2023

Raymond Chen

We saw earlier that C++ standard library shared pointers use a control block to manage the object lifetime.

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

The control block has pure virtual methods, so it is up to derived classes to establish how to dispose and delete the control block.

If you ask a `shared_ptr` to take responsibility for an already-constructed pointer, then you get this:

```
template<typename T>
struct separate_control_block : control_block
{
    virtual void Destroy() noexcept override
    {
        delete ptr;
    }
    virtual void Delete() noexcept override
    {
        delete this;
    }
    T* ptr;
};
```

Added to the basic control block is a pointer to the managed object, which is `delete`d when the last strong reference goes away.

If you use `make_shared` or `allocate_shared`, then the control block and the managed object are placed in the same allocation. In that case, the control block looks like this:

```
template<typename T>
struct combined_control_block : control_block
{
    virtual void Destroy() noexcept override
    {
        ptr()->~T();
    }
    virtual void Delete() noexcept override
    {
        delete this;
    }
    T* ptr() { return reinterpret_cast<T*>(buffer); }

    // This buffer holds a "T"
    [[alignas(T)]] char buffer[sizeof(T)];
};
```

Added to the basic control block is a buffer suitable for holding a `T` object. When the `shared_ptr` is created, a `T` is placement-constructed in that buffer, and when the last strong reference goes away (`Destroy()`), it is destructed. Stephan T. Lavavej calls this the "We know where you live" optimization because the control block doesn't need to store an explicit pointer to the buffer; it can derive it on the fly.

The reality is a little more complicated due to the need to store a deleter and possibly an allocator, but those are typically zero-length objects, so they get stored in a compressed pair with the other members.

In practice, when debugging, you don't need to look past the reference counts in the `control_block`. The thing you really care about in the `shared_ptr` is the pointed-to object. If you ever look at the control block, it's just to check whether there are any active strong references.

Bonus chatter: For C++20 `make_shared<T[]>`, there's another version of the control block that also has a `count` member which specifies how many objects are in the storage.