15.6 — std::shared_ptr

BY ALEX ON MARCH 16TH, 2017 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

Unlike std::unique_ptr, which is designed to singly own and manage a resource, std::shared_ptr is meant to solve the case where you need multiple smart pointers co-owning a resource.

This means that it is fine to have multiple std::shared_ptr pointing to the same resource. Internally, std::shared_ptr keeps track of how many std::shared_ptr are sharing the resource. As long as at least one std::shared_ptr is pointing to the resource, the resource will not be deallocated, even if individual std::shared_ptr are destroyed. As soon as the last std::shared_ptr managing the resource goes out of scope (or is reassigned to point at something else), the resource will be deallocated.

Like std::unique_ptr, std::shared_ptr lives in the <memory> header.

```
#include <iostream>
#include <memory> // for std::shared_ptr

class Resource
{
public:
    Resource() { std::cout << "Resource acquired\n"; }
    ~Resource() { std::cout << "Resource destroyed\n"; }
};

int main()
{
    // allocate a Resource object and have it owned by std::shared_ptr
    Resource *res = new Resource;
    std::shared_ptr<Resource> ptr1(res);
    {
        std::shared_ptr<Resource> ptr2(ptr1); // use copy initialization to make another std::shared_ptr pointing to the same thing

        std::cout << "Killing one shared pointer\n";
    } // ptr2 goes out of scope here, but nothing happens

    std::cout << "Killing another shared pointer\n";

    return 0;
} // ptr1 goes out of scope here, and the allocated Resource is destroyed
```

This prints:

```
Resource acquired
Killing one shared pointer
Killing another shared pointer
Resource destroyed
```

In the above code, we create a dynamic Resource object, and set a std::shared_ptr named ptr1 to manage it. Inside the nested block, we use copy initialization (which is allowed with std::shared_ptr, since the resource can be shared) to create a second std::shared_ptr (ptr2) that points to the same Resource. When ptr2 goes out of scope, the Resource is not deallocated, because ptr1 is still pointing at the Resource. When ptr1 goes out of scope, ptr1 notices there are no more std::shared_ptr managing the Resource, so it deallocates the Resource.

Note that we created a second shared pointer from the first shared pointer (using copy initialization). This is important. Consider the following similar program:

```
#include <iostream>
#include <memory> // for std::shared_ptr

class Resource
{
public:
	Resource() { std::cout << "Resource acquired\n"; }
	~Resource() { std::cout << "Resource destroyed\n"; }
};

int main()
{
	Resource *res = new Resource;
	std::shared_ptr<Resource> ptr1(res);
	{
		std::shared_ptr<Resource> ptr2(res); // create ptr2 directly from res (instead of ptr1)

		std::cout << "Killing one shared pointer\n";
	} // ptr2 goes out of scope here, and the allocated Resource is destroyed

	std::cout << "Killing another shared pointer\n";

	return 0;
} // ptr1 goes out of scope here, and the allocated Resource is destroyed again
```

This program prints:

```
Resource acquired
Killing one shared pointer
Resource destroyed
Killing another shared pointer
Resource destroyed
```

and then crashes (at least on the author's machine).

The difference here is that we created two std::shared_ptr independently from each other. As a consequence, even though they're both pointing to the same Resource, they aren't aware of each other. When ptr2 goes out of scope, it thinks it's the only owner of the Resource, and deallocates it. When ptr1 later goes out of the scope, it thinks the same thing, and tries to delete the Resource again. Then bad things happen.

Fortunately, this is easily avoided by using copy assignment or copy initialization when you need multiple shared pointers pointing to the same Resource.

Rule: Always make a copy of an existing std::shared_ptr if you need more than one std::shared_ptr pointing to the same resource.

std::make_shared

Much like std::make_unique() can be used to create a std::unique_ptr in C++14, std::make_shared() can (and should) be used to make a std::shared_ptr. std::make_shared() is available in C++11.

Here's our original example, using std::make_shared():

```
#include <iostream>
#include <memory> // for std::shared_ptr

class Resource
{
public:
```

```
    Resource() { std::cout << "Resource acquired\n"; }
    ~Resource() { std::cout << "Resource destroyed\n"; }
};

int main()
{
    // allocate a Resource object and have it owned by std::shared_ptr
    auto ptr1 = std::make_shared<Resource>();
    {
        auto ptr2 = ptr1; // create ptr2 using copy initialization of ptr1

        std::cout << "Killing one shared pointer\n";
    } // ptr2 goes out of scope here, but nothing happens

    std::cout << "Killing another shared pointer\n";

    return 0;
} // ptr1 goes out of scope here, and the allocated Resource is destroyed
```

The reasons for using std::make_shared() are the same as std::make_unique() -- std::make_shared() is simpler and safer (there's no way to directly create two std::shared_ptr pointing to the same resource using this method). However, std::make_shared() is also more performant than not using it. The reasons for this lie in the way that std::shared_ptr keeps track of how many pointers are pointing at a given resource.

Digging into std::shared_ptr

Unlike std::unique_ptr, which uses a single pointer internally, std::shared_ptr uses two pointers internally. One pointer points at the resource being managed. The other points at a "control block", which is a dynamically allocated object that tracks of a bunch of stuff, including how many std::shared_ptr are pointing at the resource. When a std::shared_ptr is created via a std::shared_ptr constructor, the memory for the managed object (which is usually passed in) and control block (which the constructor creates) are allocated separately. However, when using std::make_shared(), this can be optimized into a single memory allocation, which leads to better performance.

This also explains why independently creating two std::shared_ptr pointed to the same resource gets us into trouble. Each std::shared_ptr will have one pointer pointing at the resource. However, each std::shared_ptr will independently allocate its own control block, which will indicate that it is the only pointer owning that resource. Thus, when that std::shared_ptr goes out of scope, it will deallocate the resource, not realizing there are other std::shared_ptr also trying to manage that resource.

However, when a std::shared_ptr is cloned using copy assignment, the data in the control block can be appropriately updated to indicate that there are now additional std::shared_ptr co-managing the resource.

Shared pointers can be created from unique pointers

A std::unique_ptr can be converted into a std::shared_ptr via a special std::shared_ptr constructor that accepts a std::unique_ptr r-value. The contents of the std::unique_ptr will be moved to the std::shared_ptr.

However, std::shared_ptr can not be safely converted to a std::unique_ptr. This means that if you're creating a function that is going to return a smart pointer, you're better off returning a std::unique_ptr and assigning it to a std::shared_ptr if and when that's appropriate.

The perils of std::shared_ptr

std::shared_ptr has some of the same challenges as std::unique_ptr -- if the std::shared_ptr is not properly disposed of (either because it was dynamically allocated and never deleted, or it was part of an object that was dynamically allocated and never deleted) then the resource it is managing won't be deallocated either. With std::unique_ptr, you only have to worry about one smart pointer being properly disposed of. With std::shared_ptr, you have to worry about them all. If any of the std::shared_ptr managing a resource are not properly destroyed, the resource will not be deallocated properly.

std::shared_ptr and arrays

In C++14 and earlier, std::shared_ptr does not have proper support for managing arrays, and should not be used to manage a C-style array. As of C++17, std::shared_ptr does have support for arrays. However, as of C++17, std::make_shared is still lacking proper support for arrays, and should not be used to create shared arrays. This will likely be addressed in C++20.

Conclusion

std::shared_ptr is designed for the case where you need multiple smart pointers co-managing the same resource. The resource will be deallocated when the last std::shared_ptr managing the resource is destroyed.

15.7 -- Circular dependency issues with std::shared_ptr, and std::weak_ptr

Index

15.5 -- std::unique_ptr

C++ TUTORIAL | PRINT THIS POST

46 comments to 15.6 — std::shared_ptr

hellmet
January 30, 2020 at 10:23 am · Reply

Say I want to ensure I the resource in shared_ptr count goes all the way to zero at the end of my program. How can I do that?

I can't obviously check that the count is zero, since by the time it's zero, the shared_ptr is also hopefully and probably being destroyed (because shared_ptr.use_count() == 0 means that something weird is happening or make_shared wasn't used, I'm guessing). The closest I can get is to check if at the end of the expected lifetime a.k.a scope of the shared_ptr, it's use_count() == 1. This way, at the end of the scope, the shared_ptr destroys along with it the shared_ptr, and the resource it holds, correct?

Is there any better way?

nascardriver
January 31, 2020 at 12:36 am · Reply

A \`std::weak_ptr\` can share a resource with a \`std::shared_ptr\` without owning it, so you can check if the use count is 0 at some point. That only works if you know the point at which the resource should have been freed.
If you don't know how long the resource will live for, you can use use valgrind to check for memory leaks.
Since a \`std::shared_ptr\` that doesn't die will leak memory, valgrind will catch it.

hellmet