22.6 — std::shared_ptr

ALEX[1] JUNE 2, 2024

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 }; // 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 the copy constructor 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. 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: if you need more than one std::shared_ptr to a given resource, copy an existing std::shared_ptr.

Best practice

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

Just like with std::unique_ptr, std::shared_ptr can be a null pointer, so check to make sure it is valid before using it.

---

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 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 create two independent std::shared_ptr pointing to the same resource but unaware of each other 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++17 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++20, std::shared_ptr does have support for arrays.

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.

Next lesson

22.7 Circular dependency issues with std::shared_ptr, and std::weak_ptr

2

Back to table of contents

3

Previous lesson

22.5 std::unique_ptr

4

5

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[6] are connected to your provided email address.

119 COMMENTS

Newest

LuluBep

April 26, 2025 1:28 pm PDT

Dear Alex,
In the line : `Just like with std::unique_ptr, std::shared_ptr can be a null pointer, so check to make sure it is valid before using it.` : I understand as std::shared_ptr is also has an implicit conversion such as std::unique_ptr (the syntax should be: `if (std::shared_ptr)`. Am I correct?

0 Reply

NordicCat

January 15, 2025 9:54 pm PST

```cpp
// Creating the party and the first guest signs in
std::shared_ptr<Resource> mainGuest(new Resource());  // Reference count = 1

{
    // Another guest arrives and signs the same guest book
    std::shared_ptr<Resource> newGuest(mainGuest);    // Reference count = 2

    // newGuest leaves and signs out
}  // Reference count back to 1

// mainGuest leaves, sees they're the last one (count = 0)
// and turns off the lights, cleans up
```

nice example!

2 Reply

Cpp Learner

January 10, 2025 7:12 pm PST

As example you should instead use this

```
#include <iostream>
#include <memory> // for std::unique_ptr
#include <utility> // for std::move

class Resource
{
public:
    int Test = 0;

    explicit Resource(const int test)
    {
        Test = test;
        std::cout << "constructed:" << Test << "\n";
    }

    ~Resource() { std::cout << "destructed:" << Test << "\n"; }
};

int main()
{
    std::cout << std::unitbuf;
    Resource* res{new Resource{0}};
    std::shared_ptr<Resource> ptr1{res};
    {
        std::shared_ptr<Resource> ptr2{res};
        std::cout << "Destructing" << std::endl;
    }

    return 0;
} // Resource destroyed here when res2 goes out of scope
```

Once one of shared_ptr destructing, we cannot know how other two affected from your example. However when one shared_ptr destructed I can see clearly from the member variable, object is destructed and member values are dangling. Also this is a good example for dangling pointers

0 Reply

EmtyC

December 21, 2024 8:21 am PST

I have a thought:
Why doesn't std::shared_ptr keep a static list of addresses it points to currently (between each template instantiation aka same type std::shared_ptr, or a global variable between all template instantiations) that way, we may have

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

work correctly if I am not wrong

1 Reply

Alex Author

Reply to EmtyC [7] December 29, 2024 8:14 pm PST

I'm not sure. It could be an efficiency issue (having global data would require some kind of hash), or maybe a threading issue since you could have multiple threads modifying the global ownership data simultaneously.

2 Reply

EmtyC

Reply to Alex [8] December 29, 2024 10:34 pm PST

hmmm, didn't thought of multithreading, tnx :>

1 Reply

Nitin

November 11, 2024 9:27 pm PST

Hi Alex, seeking your expertise over this below code snippet. Thank you!

```
1 auto ptr1 { std::make_shared<Resource>() };
2 auto ptr2 { std::make_shared<Resource>() };
```

Here ptr1 and ptr2 would be pointing to two entirely independent Resource objects right? They aren't sharing anything here, and neither are they aware of one another. Hope I'm not wrong.

1 Reply

Alex Author

Reply to Nitin [9] November 14, 2024 1:38 pm PST

Correct.

3 Reply

Asgar

November 10, 2024 3:43 pm PST

Hi Alex,

Something about this text:
"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."

If the counter information too is dynamically allocated, then how will weak_ptr's expired() method work, which returns true if the counter == 0? We know, the last of a group of shared pointers deallocates everything that the first shared pointer allocated. Which means, the counter info too!

0 Reply

Alex Author

Reply to Asgar [10] November 14, 2024 8:51 am PST

It's implementation defined. One possible implementation is to have the control block keeps track of both how many `shared_ptr` and how many `weak_ptr` are pointed at it, and doesn't deallocate until both are 0. In this implementation, when the last `shared_ptr` is gone, the resource being managed is deleted, but the control block will stick around if any `weak_ptr` are still outstanding.

0 Reply

Phargelm

August 21, 2024 4:06 pm PDT

Is there any suggestions on how we should pass shared pointers to functions. Are the advices from the previous lesson (section "Passing std::unique_ptr to a function") are the same for `std::shared_ptr`?

0 Reply

Alex Author

Reply to Phargelm [11] August 22, 2024 2:08 pm PDT

Same advice. Pass the object being managed if the function only needs to access the object. Pass the whole shared_ptr if the function is involved in a possible change of ownership.

2 Reply

Karl

June 1, 2024 3:57 am PDT

In the std::make_shared section you state:
> "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)."

Isn't the whole point of `shared_ptr` to have multiple pointers pointing to the same resource? I'm not sure I understand what you're saying here.

0 Reply

Karl

Reply to Karl [12] June 1, 2024 4:28 am PDT

I just realized what you were getting at. If you initialize a `shared_ptr` with `make_shared` there is no way you can **independently** initialize two shared pointers from the same pointer, making each of the two shared pointers ignorant of the existence of the other (as illustrated in the earlier example). You could just as easily avoid this by only initializing with `std::shared_ptr<Resource> res{new`

`Resource()}`. Obviously you would miss out on the performance benefits of `std::make_shared`. Is my assessment correct or am I misinterpreting?

1 Reply

Alex Author

Reply to Karl [13] June 2, 2024 4:12 pm PDT

Correct. I rewrote the sentence to be a little more clear about this.

1 Reply

darkness

April 23, 2024 1:14 am PDT

Hi Alex, I have a question to consult, the c++11 version of shared_ptr does not support managing array resources, but I am curious why the following code works when I specify -std=c++11 (environment: g++11.3 ubuntu22.04) also compiles, btw, which can also be tested at `https://www.onlinegdb.com/`, hope you happen to know the answer.ths

```
1 std::shared_ptr<int[]> s_ptr(new int[10]());
2 s_ptr[1] = 10;
```

1 Reply

Alex Author

Reply to darkness [14] April 25, 2024 2:15 pm PDT

`new int[10]` will decay to an `int*`, and the `shared_ptr` will treat that as a pointer to a scalar rather than a pointer to an array. When deallocation time comes, scalar delete will be used.

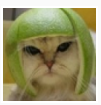
1 Reply

Erik

April 5, 2024 11:35 am PDT

I am still using this site as my primary reference site... You stress checking for nullptr if using unique_ptr, but to my understanding this also applies to shared_ptr (and weak I guess) the same way? If so, maybe a quick reminder would be helpfull :)

1 Reply



Alex Author

Reply to Erik [15] April 6, 2024 11:32 am PDT

Added a note into the lesson to make this explicit. Thanks for the feedback.

1 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/circular-dependency-issues-with-stdshared_ptr-and-stdweak_ptr/
3. https://www.learncpp.com/
4. https://www.learncpp.com/cpp-tutorial/stdunique_ptr/
5. https://www.learncpp.com/stdshared_ptr/
6. https://gravatar.com/
7. https://www.learncpp.com/cpp-tutorial/stdshared_ptr/#comment-605531
8. https://www.learncpp.com/cpp-tutorial/stdshared_ptr/#comment-605928
9. https://www.learncpp.com/cpp-tutorial/stdshared_ptr/#comment-604095
10. https://www.learncpp.com/cpp-tutorial/stdshared_ptr/#comment-604051
11. https://www.learncpp.com/cpp-tutorial/stdshared_ptr/#comment-601138
12. https://www.learncpp.com/cpp-tutorial/stdshared_ptr/#comment-597808
13. https://www.learncpp.com/cpp-tutorial/stdshared_ptr/#comment-597811
14. https://www.learncpp.com/cpp-tutorial/stdshared_ptr/#comment-596101
15. https://www.learncpp.com/cpp-tutorial/stdshared_ptr/#comment-595468
16. https://g.ezoic.net/privacy/learncpp.com