M.7 — std::shared_ptr

ALEX **OCTOBER 11, 2021**

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.

```
 1  #include <iostream>
 2  #include <memory> // for std::shared_ptr
 3
 4  class Resource
 5  {
 6  public:
 7      Resource() { std::cout << "Resource acquired\n"; }
 8      ~Resource() { std::cout << "Resource destroyed\n"; }
 9  };
10
11  int main()
12  {
13      // allocate a Resource object and have it owned by std::shared_ptr
14      Resource *res = new Resource;
15      std::shared_ptr<Resource> ptr1{ res };
16      {
17          std::shared_ptr<Resource> ptr2 { ptr1 }; // make another std::shared_ptr pointing to the same
18  thing
19
20          std::cout << "Killing one shared pointer\n";
21      } // ptr2 goes out of scope here, but nothing happens
22
23      std::cout << "Killing another shared pointer\n";
24
25      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.

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 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++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
M.8 Circular dependency issues with std::shared_ptr, and std::weak_ptr

Back to table of contents

Previous lesson
M.6 std::unique_ptr

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

Leave a comment...

Name*

Email*

Notify me about replies: POST COMMENT

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

73 COMMENTS Newest

Sarquael
December 10, 2021 10:46 am

Hi Alex.
Does std::make_shared allocate given resource dynamically? If not can we make it to do so like in std::shared_ptr?

0 Reply

> **Alex** Author
> Reply to Sarquael December 10, 2021 12:24 pm
>
> Yes, it does. Per https://en.cppreference.com/w/cpp/memory/shared_ptr/make_shared:
>
> > Constructs an object of type T and wraps it in a std::shared_ptr using args as the parameter list for the constructor of T. The object is constructed as if by the expression ::new (pv) T(std::forward(args)...)
>
> 0 Reply

Lydia
October 11, 2021 12:01 am

Hello, Alex! First of all, thank you very much for this website. It is really a valuable source of information :)

In your first example, I have noticed that you use brace-initialization for your smart pointer "ptr2" (line 17), but you do not do the same for your smart pointer "ptr1" (line 15). I was wondering if there is a particular reason for it :)

Thanks again!

0 Reply

> **Alex** Author
> Reply to Lydia October 11, 2021 6:39 pm
>
> No reason. I just missed updating it during the last edit. It's fixed now.
>
> 0 Reply

HangUp
September 28, 2021 2:27 am

1. Is the "control block" allocated even when the std::shared_ptr is created without a pointer(i.e. using the default constructor)?
2. Is the implementation/structure of the control block up to the compiler or are there set rules by the standard?

Also I think there's a small typo at "which is a dynamically allocated object that tracks of a bunch of stuff", I think it was supposed be "...keeps track of..".

0 Reply

> **Alex** Author
> Reply to HangUp September 28, 2021 5:03 pm
>
> The control block is an implementation-specific detail that may or may not actually exist in a compiler's specific implementation. You can read more here: https://en.cppreference.com/w/cpp/memory/shared_ptr
>
> 0 Reply

Saleh Sayed
September 11, 2021 6:42 am

Hello Alex,

I believe that by the term (Copy Initialization) in this article, you are referring to (Copy Constructor) .

If that's what you mean, I guess it would be better to use the term (Copy Constructor), as it impairs with the term (Copy Initialization) "introduced in lesson (1.4 — Variable assignment and initialization)"

0 Reply

> **Alex** Author
> Reply to Saleh Sayed September 13, 2021 10:40 pm
>
> Wording amended. Thanks for the suggestion.
>
> 0 Reply

Guang
July 9, 2021 6:55 am

This will likely be addressed in C++20. Has it been changed?

0 Reply

> **nascardriver** Sub-admin
> Reply to Guang July 10, 2021 6:50 am
>
> Yep, std::shared_ptr works with arrays. Thanks for pointing out the old note!
>
> 0 Reply

qwerty
May 20, 2021 9:23 am

'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.' should be in a green box.

0 Reply

Hovsep_Papoyan
December 14, 2020 4:23 pm

Nice explanation, but in the first code snippet, this is not copy initialization, it is direct initialization.

0 Reply

> **Rishi**
> Reply to Hovsep_Papoyan August 13, 2021 5:48 pm
>
> No, it's copy initialization only. It calls the copy constructor
>
> 0 Reply

frank.wang
December 8, 2020 5:37 am

@Alex
I have a question
class A{
std::shared_ptr<int> ptr(new int(5));
};
when compile , i get the error:
error: expected identifier before 'new'
std::shared_ptr<int> ptri(new int(5));
why cannot initialize it when i define at the same time
appreciate for any response -_-

0 Reply

> **nascardriver** Sub-admin
> Reply to frank.wang December 8, 2020 7:52 am
>
> You can't use direct-initialization to initialize class members at their declaration. Use list-initialization.

```
std::shared_ptr ptr{ new int{ 5 } };
```

> 0 Reply

>> **frank.wang**
>> Reply to nascardriver December 8, 2020 10:16 pm
>>
>> this may explain the question.
>> https://stackoverflow.com/questions/28696605/why-class-data-members-cant-be-initialized-by-direct-initialization-syntax
>>
>> thank you very much!
>>
>> 0 Reply

>> **frank.wang**
>> Reply to nascardriver December 8, 2020 7:58 pm
>>
>> thank you, nascardriver
>> Is list-initialization same as the Initializer List ?
>> and why we can't use direct-initialization to initialize class members at their declaration?
>>
>> 0 Reply

>>> **nascardriver** Sub-admin
>>> Reply to frank.wang December 9, 2020 8:35 am
>>>
>>> > Is list-initialization same as the Initializer List ?
>>> This is one of C++'s easy-to-mix-up corners

```
// List-initialization
int i{ 123 };

int i{ 123 };
//   ^^^^^^^ braced-init-list

int i{ 123 };
//     ^^^ initializer-list, also called "initializer list" (Can consist of multiple comma-separated elements)

class C
{
public:
  int m_i{};
  int m_j{};

  C(int i, int j)
    : m_i{ i }, m_j{ j } // Member initializer list
};

std::initializer_list l{ 1, 2, 3 }; // std::initializer_list, also called "initializer list"
```

>>> When I say "Use list-initialization", I mean "use curly braces, don't use parentheses, don't use an equals-sign"
>>>
>>> 0 Reply

>>>> **frank.wang**
>>>> Reply to nascardriver December 9, 2020 6:17 pm
>>>>
>>>> thank you very much!
>>>> It helped me a lot! -_-
>>>>
>>>> 0 Reply

Giang
November 29, 2020 4:46 am

hi, you said that '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'??? Ins'n the resource deallocated when the last std::shared_ptr managing the resource is destroyed e.g. when it's automatically destoyed when going out of scope?

0 Reply

> **nascardriver** Sub-admin
> Reply to Giang November 29, 2020 5:09 am
>
> Yes, but this statement assumes that "the std::shared_ptr is not properly disposed of".
> If the `std::shared_ptr` doesn't die, the resource doesn't die.
>
> 0 Reply

Alek
November 5, 2020 4:21 am

can you elaborate this please ?what do you mean by one allocation ? how can both pointers share one memory address ?or is it that it constructs an aggregate type like struct for both control block and our resoruce ? `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.`

thx in advance!

0 Reply

> **nascardriver** Sub-admin
> Reply to Alek November 6, 2020 8:12 am
>
> `std::make_shared` uses _placement-new_.
>
> It first allocates N bytes of memory without creating an object.
> Then it uses placement-new to create the control block (If required) and another placement-new to create the object. placement-new doesn't allocate any memory, it creates an object at a predefined address.
>
> Without `std::make_shared`, you'd
>
> 1. allocate memory for your object and create the object (Both using a single `new` call)
> 2. have `std::make_shared` allocate memory for- and create the control block.
>
> = 2 allocations
>
> 0 Reply