

M.7 — std::shared_ptr

ALEX SEPTEMBER 13, 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;
26 } // 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:

```

1 #include <iostream>
2 #include <memory> // for std::shared_ptr

3 class Resource
4 {
5 public:
6     Resource() { std::cout << "Resource acquired\n"; }
7     ~Resource() { std::cout << "Resource destroyed\n"; }
8 };

9 int main()
10 {
11     Resource *res = new Resource;
12     std::shared_ptr<Resource> ptr1(res);
13     {
14         std::shared_ptr<Resource> ptr2(res); // create ptr2 directly from res (instead of
15         ptr1)
16         std::cout << "Killing one shared pointer\n";
17     } // ptr2 goes out of scope here, and the allocated Resource is destroyed

18     std::cout << "Killing another shared pointer\n";
19     return 0;
20 } // 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()`:

```

1 #include <iostream>
2 #include <memory> // for std::shared_ptr

3 class Resource
4 {
5 public:
6     Resource() { std::cout << "Resource acquired\n"; }
7     ~Resource() { std::cout << "Resource destroyed\n"; }
8 };

9 int main()
10 {
11     // allocate a Resource object and have it owned by std::shared_ptr
12     auto ptr1 = std::make_shared<Resource>();
13     {
14         auto ptr2 = ptr1; // create ptr2 using copy initialization of ptr1

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

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

18     return 0;
19 } // 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`

Leave a comment... Put C++ code between triple-backticks (markdown style):````Your C++ code here````

 Name*

 Email* 

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

Notify me about replies:



POST COMMENT

DP N N FOUT

Newest ▼

