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

ALEX[1] JULY 22, 2024

In the previous lesson, we saw how std::shared_ptr allowed us to have multiple smart pointers co-owning the same resource. However, in certain cases, this can become problematic. Consider the following case, where the shared pointers in two separate objects each point at the other object:

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

class Person
{
    std::string m_name;
    std::shared_ptr<Person> m_partner; // initially created empty

public:

    Person(const std::string &name): m_name(name)
    {
        std::cout << m_name << " created\n";
    }
    ~Person()
    {
        std::cout << m_name << " destroyed\n";
    }

    friend bool partnerUp(std::shared_ptr<Person> &p1, std::shared_ptr<Person> &p2)
    {
        if (!p1 || !p2)
            return false;

        p1->m_partner = p2;
        p2->m_partner = p1;

        std::cout << p1->m_name << " is now partnered with " << p2->m_name << '\n';

        return true;
    }
};

int main()
{
    auto lucy { std::make_shared<Person>("Lucy") }; // create a Person named "Lucy"
    auto ricky { std::make_shared<Person>("Ricky") }; // create a Person named "Ricky"

    partnerUp(lucy, ricky); // Make "Lucy" point to "Ricky" and vice-versa

    return 0;
}
```

In the above example, we dynamically allocate two Persons, “Lucy” and “Ricky” using make_shared() (to ensure lucy and ricky are destroyed at the end of main()). Then we partner them up. This sets the std::shared_ptr inside “Lucy” to point at “Ricky”, and the std::shared_ptr inside “Ricky” to point at “Lucy”.

Shared pointers are meant to be shared, so it's fine that both the lucy shared pointer and Rick's m_partner shared pointer both point at "Lucy" (and vice-versa).

However, this program doesn't execute as expected:

```
Lucy created
Ricky created
Lucy is now partnered with Ricky
```

And that's it. No deallocations took place. Uh oh. What happened?

After partnerUp() is called, there are two shared pointers pointing to "Ricky" (ricky, and Lucy's m_partner) and two shared pointers pointing to "Lucy" (lucy, and Ricky's m_partner).

At the end of main(), the ricky shared pointer goes out of scope first. When that happens, ricky checks if there are any other shared pointers that co-own the Person "Ricky". There are (Lucy's m_partner). Because of this, it doesn't deallocate "Ricky" (if it did, then Lucy's m_partner would end up as a dangling pointer). At this point, we now have one shared pointer to "Ricky" (Lucy's m_partner) and two shared pointers to "Lucy" (lucy, and Ricky's m_partner).

Next the lucy shared pointer goes out of scope, and the same thing happens. The shared pointer lucy checks if there are any other shared pointers co-owning the Person "Lucy". There are (Ricky's m_partner), so "Lucy" isn't deallocated. At this point, there is one shared pointer to "Lucy" (Ricky's m_partner) and one shared pointer to "Ricky" (Lucy's m_partner).

Then the program ends -- and neither Person "Lucy" or "Ricky" have been deallocated! Essentially, "Lucy" ends up keeping "Ricky" from being destroyed, and "Ricky" ends up keeping "Lucy" from being destroyed.

It turns out that this can happen any time shared pointers form a circular reference.

Circular references

A **Circular reference** (also called a **cyclical reference** or a **cycle**) is a series of references where each object references the next, and the last object references back to the first, causing a referential loop. The references do not need to be actual C++ references -- they can be pointers, unique IDs, or any other means of identifying specific objects.

In the context of shared pointers, the references will be pointers.

This is exactly what we see in the case above: "Lucy" points at "Ricky", and "Ricky" points at "Lucy". With three pointers, you'd get the same thing when A points at B, B points at C, and C points at A. The practical effect of having shared pointers form a cycle is that each object ends up keeping the next object alive -- with the last object keeping the first object alive. Thus, no objects in the series can be deallocated because they all think some other object still needs it!

A reductive case

It turns out, this cyclical reference issue can even happen with a single std::shared_ptr -- a std::shared_ptr referencing the object that contains it is still a cycle (just a reductive one). Although it's fairly unlikely that this would ever happen in practice, we'll show you for additional comprehension:

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

class Resource
{
public:
    std::shared_ptr<Resource> m_ptr {}; // initially created empty

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

int main()
{
    auto ptr1 { std::make_shared<Resource>() };

    ptr1->m_ptr = ptr1; // m_ptr is now sharing the Resource that contains it

    return 0;
}
```

In the above example, when ptr1 goes out of scope, the Resource is not deallocated because the Resource's m_ptr is sharing the Resource. At that point, the only way for the Resource to be released would be to set m_ptr to something else (so nothing is sharing the Resource any longer). But we can't access m_ptr because ptr1 is out of scope, so we no longer have a way to do this. The Resource has become a memory leak.

Thus, the program prints:

```
Resource acquired
```

and that's it.

So what is std::weak_ptr for anyway?

std::weak_ptr was designed to solve the "cyclical ownership" problem described above. A std::weak_ptr is an observer -- it can observe and access the same object as a std::shared_ptr (or other std::weak_ptrs) but it is not considered an owner. Remember, when a std::shared pointer goes out of scope, it only considers whether other std::shared_ptr are co-owning the object. std::weak_ptr does not count!

Let's solve our Person-al issue using a std::weak_ptr:

```cpp
#include <iostream>
#include <memory> // for std::shared_ptr and std::weak_ptr
#include <string>

class Person
{
    std::string m_name;
    std::weak_ptr<Person> m_partner; // note: This is now a std::weak_ptr

public:

    Person(const std::string &name): m_name(name)
    {
        std::cout << m_name << " created\n";
    }
    ~Person()
    {
        std::cout << m_name << " destroyed\n";
    }

    friend bool partnerUp(std::shared_ptr<Person> &p1, std::shared_ptr<Person> &p2)
    {
        if (!p1 || !p2)
            return false;

        p1->m_partner = p2;
        p2->m_partner = p1;

        std::cout << p1->m_name << " is now partnered with " << p2->m_name << '\n';

        return true;
    }
};

int main()
{
    auto lucy { std::make_shared<Person>("Lucy") };
    auto ricky { std::make_shared<Person>("Ricky") };

    partnerUp(lucy, ricky);

    return 0;
}
```

This code behaves properly:

```
Lucy created
Ricky created
Lucy is now partnered with Ricky
Ricky destroyed
Lucy destroyed
```

Functionally, it works almost identically to the problematic example. However, now when ricky goes out of scope, it sees that there are no other std::shared_ptr pointing at “Ricky” (the std::weak_ptr from “Lucy” doesn’t count). Therefore, it will deallocate “Ricky”. The same occurs for lucy.

Using std::weak_ptr

One downside of std::weak_ptr is that std::weak_ptr are not directly usable (they have no operator->). To use a std::weak_ptr, you must first convert it into a std::shared_ptr. Then you can use the std::shared_ptr. To

convert a std::weak_ptr into a std::shared_ptr, you can use the lock() member function. Here's the above example, updated to show this off:

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

class Person
{
    std::string m_name;
    std::weak_ptr<Person> m_partner; // note: This is now a std::weak_ptr

public:

    Person(const std::string &name) : m_name(name)
    {
        std::cout << m_name << " created\n";
    }
    ~Person()
    {
        std::cout << m_name << " destroyed\n";
    }

    friend bool partnerUp(std::shared_ptr<Person> &p1, std::shared_ptr<Person> &p2)
    {
        if (!p1 || !p2)
            return false;

        p1->m_partner = p2;
        p2->m_partner = p1;

        std::cout << p1->m_name << " is now partnered with " << p2->m_name << '\n';

        return true;
    }

    std::shared_ptr<Person> getPartner() const { return m_partner.lock(); } // use
lock() to convert weak_ptr to shared_ptr
    const std::string& getName() const { return m_name; }
};

int main()
{
    auto lucy { std::make_shared<Person>("Lucy") };
    auto ricky { std::make_shared<Person>("Ricky") };

    partnerUp(lucy, ricky);

    auto partner = ricky->getPartner(); // get shared_ptr to Ricky's partner
    std::cout << ricky->getName() << "'s partner is: " << partner->getName() << '\n';

    return 0;
}
```

This prints:

```
Lucy created
Ricky created
Lucy is now partnered with Ricky
Ricky's partner is: Lucy
Ricky destroyed
Lucy destroyed
```

We don't have to worry about circular dependencies with std::shared_ptr variable "partner" since it's just a local variable inside the function. It will eventually go out of scope at the end of the function and the reference count will be decremented by 1.

Avoiding dangling pointers with std::weak_ptr

Consider the case where a normal "dumb" pointer is holding the address of some object, and then that object is destroyed. Such a pointer is dangling, and dereferencing the pointer will lead to undefined behavior. And unfortunately, there is no way for us to determine whether a pointer holding a non-null address is dangling or not. This is a large part of the reason dumb pointers are dangerous.

Because std::weak_ptr won't keep an owned resource alive, it's similarly possible for a std::weak_ptr to be left pointing to a resource that has been deallocated by a std::shared_ptr. However, std::weak_ptr has a neat trick up its sleeve -- because it has access to the reference count for an object, it can determine if it is pointing to a valid object or not! If the reference count is non-zero, the resource is still valid. If the reference count is zero, then the resource has been destroyed.

The easiest way to test whether a std::weak_ptr is valid is to use the `expired()` member function, which returns `true` if the std::weak_ptr is pointing to an invalid object, and `false` otherwise.

Here's a simple example showing this difference in behavior:

```
1  // h/t to reader Waldo for an early version of this example
2  #include <iostream>
3  #include <memory>
4
5  class Resource
6  {
7  public:
8      Resource() { std::cerr << "Resource acquired\n"; }
9      ~Resource() { std::cerr << "Resource destroyed\n"; }
10 };
11
12 // Returns a std::weak_ptr to an invalid object
13 std::weak_ptr<Resource> getWeakPtr()
14 {
15     auto ptr{ std::make_shared<Resource>() };
16     return std::weak_ptr<Resource>{ ptr };
17 } // ptr goes out of scope, Resource destroyed
18
19 // Returns a dumb pointer to an invalid object
20 Resource* getDumbPtr()
21 {
22     auto ptr{ std::make_unique<Resource>() };
23     return ptr.get();
24 } // ptr goes out of scope, Resource destroyed
25
26 int main()
27 {
28     auto dumb{ getDumbPtr() };
29     std::cout << "Our dumb ptr is: " << ((dumb == nullptr) ? "nullptr\n" : "non-
30 null\n");
31
32     auto weak{ getWeakPtr() };
33     std::cout << "Our weak ptr is: " << ((weak.expired()) ? "expired\n" : "valid\n");
34
35     return 0;
   }
```

This prints:

```
Resource acquired
Resource destroyed
Our dumb ptr is: non-null
Resource acquired
Resource destroyed
Our weak ptr is: expired
```

Both `getDumbPtr()` and `getWeakPtr()` use a smart pointer to allocate a Resource -- this smart pointer ensures that the allocated Resource will be destroyed at the end of the function. When `getDumbPtr()` returns a Resource*, it returns a dangling pointer (because std::unique_ptr destroyed the Resource at the end of the function). When `getWeakPtr()` returns a std::weak_ptr, that std::weak_ptr is similarly pointing to an invalid object (because std::shared_ptr destroyed the Resource at the end of the function).

Inside main(), we first test whether the returned dumb pointer is `nullptr`. Because the dumb pointer is still holding the address of the deallocated resource, this test fails. There is no way for `main()` to tell whether this pointer is dangling or not. In this case, because it is a dangling pointer, if we were to dereference this pointer, undefined behavior would result.

Next, we test whether `weak.expired()` is `true`. Because the reference count for the object being pointed to by `weak` is `0` (because the object being pointed to was already destroyed), this resolves to `true`. The code in `main()` can thus tell that `weak` is pointing to an invalid object, and we can conditionalize our code as appropriate!

Note that if a std::weak_ptr is expired, then we shouldn't call `lock()` on it, because the object being pointed to has already been destroyed, so there is no object to share. If you do call `lock()` on an expired std::weak_ptr, it will return a std::shared_ptr to `nullptr`.

Conclusion

std::shared_ptr can be used when you need multiple smart pointers that can co-own a resource. The resource will be deallocated when the last std::shared_ptr goes out of scope. std::weak_ptr can be used when you want a smart pointer that can see and use a shared resource, but does not participate in the ownership of that resource.

Quiz time

Question #1

1. Fix the program presented in the section "A reductive case" so that the Resource is properly deallocated. Do not alter the code in `main()`.

Here is the program again for ease of reference:

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

class Resource
{
public:
    std::shared_ptr<Resource> m_ptr {}; // initially created empty

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

int main()
{
    auto ptr1 { std::make_shared<Resource>() };

    ptr1->m_ptr = ptr1; // m_ptr is now sharing the Resource that contains it

    return 0;
}
```

Show Solution (javascript:void(0))[2]

Next lesson
22.x Chapter 22 summary and quiz

3

Back to table of contents

4

Previous lesson
22.6 std::shared_ptr

5

6

Avatars from https://gravatar.com/[7] are connected to your provided email address.

134 COMMENTS Newest

Manas Ghosh
May 17, 2025 11:12 am PDT

though trivial but one point to mention. `std::weak_ptr` being a view doesn't destroy or deallocate the object it is pointing to. if it does then with curcular reference it will cause to happen `delete` attempt of already deleted heap memory which is undefined behaviour and crashes the program.

0 Reply

emme
September 5, 2024 7:36 am PDT

So, from perspective of functionality, what is the difference between raw pointer and std::weak_ptr?

2 Reply

Alex Author
Reply to emme [8] September 6, 2024 12:16 pm PDT

A `std::weak_ptr` is always associated with a `std::shared_ptr`. If the pointed-to object is deleted, a raw ptr doesn't have any idea. A `std::weak_ptr` does (and can tell you via the `expired()` member function or conversion to an empty `std::shared_ptr` via `lock()`)

4 Reply

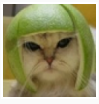
Moon
July 21, 2024 2:00 am PDT

```
const std::shared_ptr<Person> getPartner() const { return m_partner.lock(); }
```

Is there any reason why the return type of member function declaration is const value instead of non-const? When it comes to return type of function i don't know what the difference between return by value and return by const value is.

1. std::shared_ptr<Person> getPartner() const { return m_partner.lock(); }
2. const std::shared_ptr<Person> getPartner() const { return m_partner.lock(); }

1 Reply



Alex Author
Reply to Moon [9] July 22, 2024 2:21 pm PDT

The const on a by-value return type is ignored. I think it snuck in there accidentally and nobody noticed. :) I've now removed it.

0 Reply

Bohdan

July 4, 2024 8:58 am PDT

I have a question with examples (https://gist.github.com/pepsi1k/121457c690dd82ace96732808ada4b9f)[10] that would clutter comments on this site.
https://gist.github.com/pepsi1k/121457c690dd82ace96732808ada4b9f

In short, why `Resource::m_ptr` or `Person:m_partner` can not just call their destructors?

Last edited 11 months ago by Bohdan

0 Reply

Alex Author

Reply to Bohdan [11] July 7, 2024 10:14 pm PDT

Because the objects containing them never get deallocated. Using the Person example, we create two Person objects ("Ricky" and "Lucy") dynamically. We have a shared_ptr managing each (`lucy` and `ricky`). Within each, we have a `m_partner` shared_ptr pointing to the other. This means the control blocks for "Ricky" and "Lucy" each have a reference count of 2.

When main() ends, the `lucy` and `ricky` shared_ptr go out of scope. This calls the shared_ptr destructor, which decrements the reference count of each to 1. Then the destructor checks if the reference count is 0. Since it is not (because the m_partner shared_ptr is still pointing to the partner's Person), the "Lucy" and "Ricky" objects are not deallocated. And since they are not deallocated, the m_partner shared_ptr never gets destroyed, which is the thing that would decrement the reference count to 0 and actually deallocate the "Lucy" and "Ricky" person.

If we were to manually unlink the m_partner of each prior to lucy and ricky going out of scope, then things would be okay.

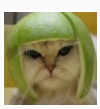
1 Reply

Bohdan

Reply to Alex [12] July 8, 2024 4:33 am PDT

Got it!
The key point is that `Person:m_partner` was created on heap by `new Person`, therefore, it can not be destroyed until `delete Person`

0 Reply



Alex Author

Reply to Bohdan [13] July 11, 2024 2:42 pm PDT

Yep!

0 Reply

Faizan

June 24, 2024 5:46 am PDT

Perhaps the easiest Quiz yet.

Overall a really good chapter. I do think I need to go over the move semantics lessons again to fully comprehend it but I get the basic gist of it.

1 Reply

JJag

May 10, 2024 11:06 am PDT

Hi Alex. Thanks for another great lesson!

I have a question regarding `weak_ptr::expired`.
How does weak pointer know if the referenced object has been deallocated?

At first I thought that `weak_ptr` just holds a pointer to a `shared_ptr`'s reference counter. However, if all `shared_ptr`s co-owning the resource would go out of scope, this reference counter would need to be deallocated along with the resource or we'd end up with a memory leak.
We could try to band-aid this issue by making `weak_pointer` actually co-own the reference counter(by having a member like `shared_ptr<int> m_refCount`),
but this still results in a ref-counter leak in a case of circular dependency - which is what we try to avoid by using `weak_ptr` in the first place.

Cheers!

0 Reply

JJag

Reply to JJag [14] May 11, 2024 4:33 am PDT

After giving it a second thought in the morning, seems it Allegro should be ok if weak pointer would co-own the counter. The counter will be alive as long as there is any shared or weak pointer alive, but since the circular dependent objects can nów be properly deallocated, their weak pointers will be destroyed as well and the counter can be freed.

0 Reply

Alex Author

Reply to JJag [15] May 11, 2024 2:56 pm PDT

Yes, the counter block is co-owned by the shared_ptr and weak_ptr.

1 Reply

Dongbin
March 6, 2024 8:07 am PST

To check my understanding, in the example at the beginning of the lesson, the dynamically allocated Person instances do not "go out of scope" at the end of the program because they were dynamically allocated?

0 Reply

Alex Author
Reply to Dongbin [16] March 8, 2024 10:33 pm PST

Correct. They aren't deallocated because dynamic memory needs to be explicitly deleted.

The OS will clean up the memory leak when the process is shut down.

0 Reply

D D
December 1, 2023 9:24 pm PST

Hello.

1. For what do we use weak PTR in the class Person, why don't we pass to the partner up just raw pointers, but do it with shared ones?
2. Why our private field isn't a raw pointer. Which sense of smart one?

Last edited 1 year ago by D D

0 Reply

Alex Author
Reply to D D [17] December 3, 2023 1:53 pm PST

`std::weak_ptr` allows us to view a `shared_ptr`, but is a non-owning pointer so it won't prevent destruction. The big advantage of `std::weak_ptr` is that it can check whether the object it is referencing is expired (destroyed). A raw pointer can't do that.

0 Reply

jason
August 7, 2023 2:44 am PDT

at **Avoiding dangling pointers with std::weak_ptr** section

line 16 should be

```
1 | return std::weak_ptr<Resource>{ ptr };
```

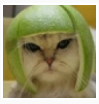
but currently is

```
1 | return std::weak_ptr{ ptr };
```

which will lead a compiling error

```
smart_pointer_09.cpp:16:29: error: missing template arguments before '{' token
   16 |         return std::weak_ptr{ ptr };
      |                             ^
```

0 Reply



Alex Author

Reply to jason [18] August 11, 2023 10:15 am PDT

It actually compiles as written in C++17 (where it uses CTAD -- class template argument deduction). But I added the explicit type argument so that it will compile in C++14 too.

1 Reply

Jason

Reply to Alex [19] August 12, 2023 9:58 am PDT

Ok, Thank you, I understand.

0 Reply

learnccp lesson reviewer

August 1, 2023 6:54 am PDT

what about a lesson/thread for general discussion like in here like a reddit where people can post question messages, search, kinda a forum at top of page

0 Reply

learnccp lesson reviewer

Reply to learnccp lesson reviewer [20] August 1, 2023 6:55 am PDT

within this page

0 Reply

learnccp lesson reviewer

Reply to learnccp lesson reviewer [21] August 1, 2023 6:55 am PDT

wEbSiItE

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. javascript:void(0)
3. https://www.learncpp.com/cpp-tutorial/chapter-22-summary-and-quiz/
4. https://www.learncpp.com/
5. https://www.learncpp.com/cpp-tutorial/stdshared_ptr/
6. https://www.learncpp.com/circular-dependency-issues-with-stdshared_ptr-and-stdweak_ptr/
7. https://gravatar.com/
8. https://www.learncpp.com/cpp-tutorial/circular-dependency-issues-with-stdshared_ptr-and-stdweak_ptr/#comment-601658
9. https://www.learncpp.com/cpp-tutorial/circular-dependency-issues-with-stdshared_ptr-and-stdweak_ptr/#comment-599901
10. https://gist.github.com/pepsi1k/121457c690dd82ace96732808ada4b9f
11. https://www.learncpp.com/cpp-tutorial/circular-dependency-issues-with-stdshared_ptr-and-stdweak_ptr/#comment-599186
12. https://www.learncpp.com/cpp-tutorial/circular-dependency-issues-with-stdshared_ptr-and-stdweak_ptr/#comment-599304
13. https://www.learncpp.com/cpp-tutorial/circular-dependency-issues-with-stdshared_ptr-and-stdweak_ptr/#comment-599310
14. https://www.learncpp.com/cpp-tutorial/circular-dependency-issues-with-stdshared_ptr-and-stdweak_ptr/#comment-596930
15. https://www.learncpp.com/cpp-tutorial/circular-dependency-issues-with-stdshared_ptr-and-stdweak_ptr/#comment-596946
16. https://www.learncpp.com/cpp-tutorial/circular-dependency-issues-with-stdshared_ptr-and-stdweak_ptr/#comment-594363
17. https://www.learncpp.com/cpp-tutorial/circular-dependency-issues-with-stdshared_ptr-and-stdweak_ptr/#comment-590450
18. https://www.learncpp.com/cpp-tutorial/circular-dependency-issues-with-stdshared_ptr-and-stdweak_ptr/#comment-585219
19. https://www.learncpp.com/cpp-tutorial/circular-dependency-issues-with-stdshared_ptr-and-stdweak_ptr/#comment-585392
20. https://www.learncpp.com/cpp-tutorial/circular-dependency-issues-with-stdshared_ptr-and-stdweak_ptr/#comment-584866
21. https://www.learncpp.com/cpp-tutorial/circular-dependency-issues-with-stdshared_ptr-and-stdweak_ptr/#comment-584867
22. https://g.ezoic.net/privacy/learncpp.com