

## 15.7 — Circular dependency issues with std::shared\_ptr, and std::weak\_ptr

BY ALEX ON MARCH 21ST, 2017 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

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:

```

1  #include <iostream>
2  #include <memory> // for std::shared_ptr
3  #include <string>
4
5  class Person
6  {
7      std::string m_name;
8      std::shared_ptr<Person> m_partner; // initially created empty
9
10 public:
11
12     Person(const std::string &name): m_name(name)
13     {
14         std::cout << m_name << " created\n";
15     }
16     ~Person()
17     {
18         std::cout << m_name << " destroyed\n";
19     }
20
21     friend bool partnerUp(std::shared_ptr<Person> &p1, std::shared_ptr<Person> &p2)
22     {
23         if (!p1 || !p2)
24             return false;
25
26         p1->m_partner = p2;
27         p2->m_partner = p1;
28
29         std::cout << p1->m_name << " is now partnered with " << p2->m_name << "\n";
30
31         return true;
32     }
33 };
34
35 int main()
36 {
37     auto lucy = std::make_shared<Person>("Lucy"); // create a Person named "Lucy"
38     auto ricky = std::make_shared<Person>("Ricky"); // create a Person named "Ricky"
39
40     partnerUp(lucy, ricky); // Make "Lucy" point to "Ricky" and vice-versa
41
42     return 0;
43 }
```

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 the function, 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:

```
1  #include <iostream>
2  #include <memory> // for std::shared_ptr
3
4  class Resource
5  {
6  public:
7      std::shared_ptr<Resource> m_ptr; // initially created empty
8
9      Resource() { std::cout << "Resource acquired\n"; }
10     ~Resource() { std::cout << "Resource destroyed\n"; }
11 };
```

```

12
13 int main()
14 {
15     auto ptr1 = std::make_shared<Resource>();
16
17     ptr1->m_ptr = ptr1; // m_ptr is now sharing the Resource that contains it
18
19     return 0;
20 }

```

In the above example, when ptr1 goes out of scope, it doesn't deallocate the Resource because the Resource's m\_ptr is sharing the Resource. Then there's nobody left to delete the Resource (m\_ptr never goes out of scope, so it never gets a chance). 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:

```

1  #include <iostream>
2  #include <memory> // for std::shared_ptr and std::weak_ptr
3  #include <string>
4
5  class Person
6  {
7      std::string m_name;
8      std::weak_ptr<Person> m_partner; // note: This is now a std::weak_ptr
9
10 public:
11
12     Person(const std::string &name): m_name(name)
13     {
14         std::cout << m_name << " created\n";
15     }
16     ~Person()
17     {
18         std::cout << m_name << " destroyed\n";
19     }
20
21     friend bool partnerUp(std::shared_ptr<Person> &p1, std::shared_ptr<Person> &p2)
22     {
23         if (!p1 || !p2)
24             return false;
25
26         p1->m_partner = p2;
27         p2->m_partner = p1;
28
29         std::cout << p1->m_name << " is now partnered with " << p2->m_name << "\n";
30
31         return true;
32     }
33 };
34

```

```

35  int main()
36  {
37      auto lucy = std::make_shared<Person>("Lucy");
38      auto ricky = std::make_shared<Person>("Ricky");
39
40      partnerUp(lucy, ricky);
41
42      return 0;
43  }

```

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

The 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:

```

1  #include <iostream>
2  #include <memory> // for std::shared_ptr and std::weak_ptr
3  #include <string>
4
5  class Person
6  {
7      std::string m_name;
8      std::weak_ptr<Person> m_partner; // note: This is now a std::weak_ptr
9
10 public:
11
12     Person(const std::string &name) : m_name(name)
13     {
14         std::cout << m_name << " created\n";
15     }
16     ~Person()
17     {
18         std::cout << m_name << " destroyed\n";
19     }
20
21     friend bool partnerUp(std::shared_ptr<Person> &p1, std::shared_ptr<Person> &p2)
22     {
23         if (!p1 || !p2)
24             return false;
25
26         p1->m_partner = p2;
27         p2->m_partner = p1;
28
29         std::cout << p1->m_name << " is now partnered with " << p2->m_name << "\n";
30
31         return true;

```

```

32     }
33
34     const std::shared_ptr<Person> getPartner() const { return m_partner.lock(); } // use lock
    () to convert weak_ptr to shared_ptr
35     const std::string& getName() const { return m_name; }
36 };
37
38 int main()
39 {
40     auto lucy = std::make_shared<Person>("Lucy");
41     auto ricky = std::make_shared<Person>("Ricky");
42
43     partnerUp(lucy, ricky);
44
45     auto partner = ricky->getPartner(); // get shared_ptr to Ricky's partner
46     std::cout << ricky->getName() << "'s partner is: " << partner->getName() << '\n';
47
48     return 0;
49 }

```

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.

## 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

1) Fix the "reductive case" program so that the Resource is properly deallocated.

## Show Solution



**15.x -- Chapter 15 comprehensive review**



**Index**



**15.6 -- std::shared\_ptr**