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 24TH, 2018

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:

```cpp
#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, the ricky shared pointer goes out of scope. When that happens, ricky checks if there are any other shared pointers co-own the Person “Ricky”. There are (the shared pointer inside Person “Lucy”). Because of this, it doesn’t deallocate “Ricky” (if it did, then “Lucy” would end up with a hanging pointer). Next lucy 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 (the shared pointer inside Person “Ricky”). So it doesn’t deallocate Lucy.

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 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:

```
#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, 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:

```
#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

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:

```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;
    }

    const 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.

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.

Hide Solution

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

class Resource
{
public:
    std::weak_ptr<Resource> m_ptr; // use std::weak_ptr so m_ptr doesn't keep the Resource alive

    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;
}
```

15.x -- Chapter 15 comprehensive review

Index

15.6 -- std::shared_ptr

27 comments to 15.7 — Circular dependency issues with std::shared_ptr, and std::weak_ptr

MarvinBruh
February 7, 2018 at 2:57 am · Reply