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

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:

```
#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:

```
#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;
	}

	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.

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:

```
// h/t to reader Waldo for an early version of this example
#include <iostream>
#include <memory>

class Resource
{
public:
        Resource() { std::cerr << "Resource acquired\n"; }
        ~Resource() { std::cerr << "Resource destroyed\n"; }
};

// Returns a std::weak_ptr to an invalid object
std::weak_ptr<Resource> getWeakPtr()
{
        auto ptr{ std::make_shared<Resource>() };
        return std::weak_ptr<Resource>{ ptr };
} // ptr goes out of scope, Resource destroyed

// Returns a dumb pointer to an invalid object
Resource* getDumbPtr()
{
        auto ptr{ std::make_unique<Resource>() };
        return ptr.get();
} // ptr goes out of scope, Resource destroyed

int main()
{
        auto dumb{ getDumbPtr() };
        std::cout << "Our dumb ptr is: " << ((dumb == nullptr) ? "nullptr\n" : "non-
null\n");

        auto weak{ getWeakPtr() };
        std::cout << "Our weak ptr is: " << ((weak.expired()) ? "expired\n" :
"valid\n");

        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:

```
#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