8、std::shared_ptr的循环依赖问题，以及std::weak_ptr

Chapter	M
Name	xingzp

在上一课中，我们看到std::shared_ptr如何允许多个智能指针共同拥有同一个资源。然而，在某些情况下，这可能会产生问题。考虑以下情况，两个单独对象中的共享指针相互指向另一个对象:

```
#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;
}
```

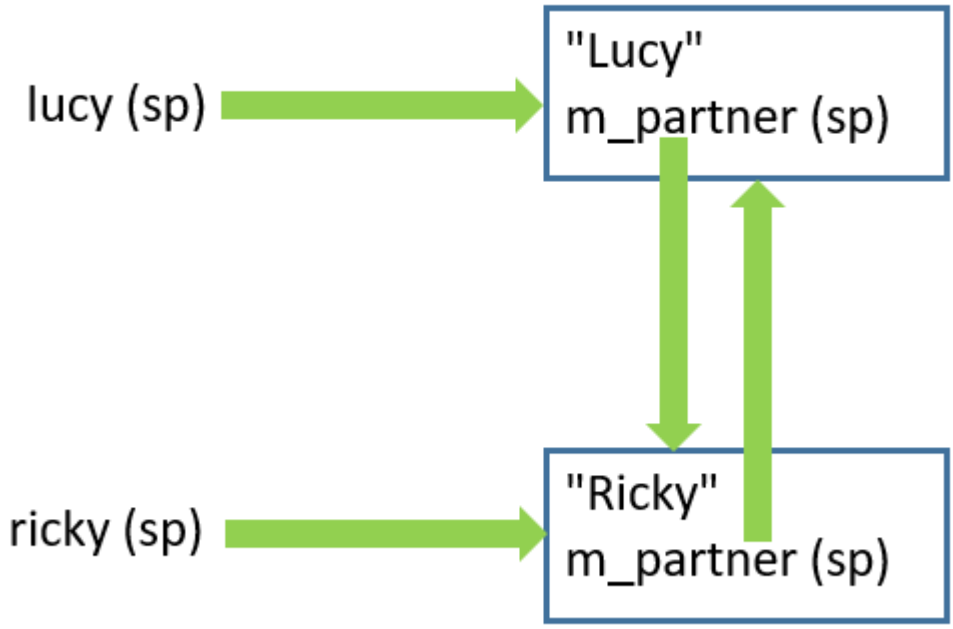
在上面的例子中，我们使用make_shared()动态分配了两个person，“Lucy”和“Ricky”(以确保lucy和ricky在main()末尾被销毁)。然后我们让他们搭档。这将“Lucy”内部的std::shared_ptr设置为指向“Ricky”，而“Ricky”内部的std::shared_ptr设置为指向“Lucy”。共享指针是用来共享的，所以lucy共享指针和Ricky的m_partner共享指针都指向“Lucy”是没有问题的(反之亦然)。

然而，这个程序并没有按照预期执行:

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

就是这样。对象没有被销毁。哦哦。发生了什么事?

在调用partnerUp()之后，有两个指向“Ricky”的共享指针(ricky，和Lucy的m_partner)和两个指向“Lucy”的共享指针(lucy，和Ricky的m_partner)。



在main()结束时，ricky共享指针首先超出作用域。当发生这种情况时，ricky检查是否有其他共同拥有“Ricky”的共享指针。有(Lucy的m_partner)。因此，它不会释放“Ricky”(如果它释放了，那么Lucy的m_partner将最终成为一个悬浮指针)。此时，我们现在有一个共享指针指向" Ricky " (Lucy的m_partner)和两个共享指针指向" Lucy " (Lucy，和Ricky的m_partner)。

接下来lucy共享指针超出作用域，同样的事情会发生。共享指针lucy检查是否有其他共享指针共同拥有" Lucy "。有(Ricky的m_partner)，所以“Lucy”没有被解除。此时，有一个共享指针指向“Lucy”(Ricky的m_partner)和一个共享指针指向“Ricky”(Lucy的m_partner)。

然后程序结束了——“Lucy”和“Ricky”都没有被释放!从本质上说，“Lucy”最终使“Ricky”免于毁灭，而“Ricky”最终使“Lucy”免于毁灭。

事实证明，在共享指针形成循环引用的任何时候都可能发生这种情况。

循环引用

循环引用(**Circular reference、也可叫做cyclical reference或者cycle**)是一系列引用：其中每个对象引用下一个对象，最后一个对象引用回第一个对象。引用不需要是实际的c++引用——它们可以是指针、惟一id或任何其他标识特定对象的方法。

在共享指针的上下文中，**循环引用**用的是指针。

这正是我们在上面的例子中看到的:“Lucy”指向“Ricky”，而“Ricky”指向“Lucy”。三个指针的情况下，当A指向B, B指向C, C指向A时，你会得到同样的结果。共享指针形成一个循环的实际效果是，每个对象最终使下一个对象保持活动——最后一个对象使第一个对象保持活动。因此，系列中的任何对象都不能被释放，因为它们都认为其他对象仍然需要它!

更简化的case

事实证明，这种循环引用问题甚至可以发生在单个std::shared_ptr中——一个std::shared_ptr指向一个对象但是该对象包含这个std::shared_ptr，这仍然是一个循环(只是一个简化版)。尽管这在实践中是不太可能发生的，但我们将向你展示更多的理解:

```
#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;
}
```

在上面的例子中，当ptr1超出作用域时，资源不会被释放，因为资源的m_ptr正在共享资源。在这一点上，释放资源的唯一方法是将m_ptr设置为其他东西(这样的话就没有任何东西再共享该资源)。但是我们已经不能访问m_ptr了，因为ptr1已经超出了作用域，于是便发生了内存泄漏。

因此，程序输出:

```
Resource acquired
```

那么std::weak_ptr到底是干什么用的呢?

std::weak_ptr被设计用来解决上面描述的“周期性所有权”问题。std::weak_ptr是一个观察者——它可以观察和访问与std::shared_ptr(或其他std::weak_ptrs)相同的对象，但它不被视为所有者。记住，当std::shared指针超出作用域时，它只考虑其他std::shared_ptr是否共同拥有该对象。std::weak_ptr不用考虑在其中!

让我们使用std::weak_ptr来解决person问题:

```
#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;
}
```

上述程序打印：

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

从功能上讲，它的工作原理与问题示例几乎相同。然而，现在当ricky超出作用域时，它会看到没有其他std::shared_ptr指向“Ricky”(来自“Lucy”的std::weak_ptr不计算)。因此，它将释放“Ricky”。lucy也是如此。

使用std::weak_ptr

std::weak_ptr的缺点是std::weak_ptr不能直接使用(它们没有操作符->)。要使用std::weak_ptr，必须首先将其转换为std::shared_ptr。然后可以使用std::shared_ptr。要将std::weak_ptr转换为std::shared_ptr，可以使用lock()成员函数。拿上面的例子稍作修改:

```
#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;
}
```

上述程序打印：

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

我们不必担心std::shared_ptr变量“partner”的循环依赖关系，因为它只是函数中的一个局部变量。它最终将在函数结束时超出作用域，引用计数将减少1。

---

std::weak_ptr悬空指针

因为std::weak_ptr不会保留一个拥有的资源，所以std::weak_ptr可能会指向一个已经被std::shared_ptr释放的资源。这样的std::weak_ptr是悬空的，使用它将导致未定义的行为。

这里有一个简单的例子来说明这是如何发生的:

```
#include <iostream>
#include <memory>

class Resource
{
```

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

auto getWeakPtr()
{
  auto ptr{ std::make_shared<Resource>() }; // Resource acquired

  return std::weak_ptr{ ptr };
} // ptr goes out of scope, Resource destroyed

int main()
{
  std::cerr << "Getting weak_ptr...\n";

  auto ptr{ getWeakPtr() }; // dangling

  std::cerr << "Done.\n";
}
```

在上面的例子中，在getWeakPtr()内部，我们使用std::make_shared()来创建一个名为ptr的std::shared_ptr变量，它拥有一个Resource对象。该函数将std::weak_ptr返回给调用者，它不增加引用计数。然后，因为ptr是一个局部变量，它在函数结束时超出了作用域，从而将引用计数减为0并释放Resource对象。返回的std::weak_ptr是悬空的，指向一个被释放的资源。

结论

当您需要多个智能指针共同拥有一个资源时，可以使用std::shared_ptr。当最后一个std::shared_ptr超出作用域时，资源将被释放。当您想要一个可以看到和使用共享资源，但不参与该资源所有权的智能指针时，可以使用std::weak_ptr。