6、std::unique_ptr

Chapter	M
Name	xingzp

在本章的开头，我们讨论了在某些情况下使用指针如何导致错误和内存泄漏。例如，当函数提前返回或抛出异常，而指针没有被正确删除时，就会发生这种情况。

```
#include <iostream>

void someFunction()
{
    auto* ptr{ new Resource() };

    int x{};
    std::cout << "Enter an integer: ";
    std::cin >> x;

    if (x == 0)
        throw 0; // the function returns early, and ptr won't be deleted!

    // do stuff with ptr here

    delete ptr;
}
```

现在我们已经介绍了移动语义的基础知识，我们可以回到智能指针类的主题。提醒一下，智能指针是一个管理动态分配对象的类。尽管智能指针可以提供其他功能，但智

能指针的定义特征是它管理动态分配的资源，并确保在适当的时间(通常是在智能指针超出作用域时)正确清理动态分配的对象。

正因为如此，**智能指针永远不应该自己动态分配**(否则，智能指针可能没有被正确地释放，这意味着它所拥有的对象将没有被释放，导致内存泄漏)。通过始终在堆栈上分配智能指针(作为局部变量或类的组合成员)，我们可以保证智能指针在包含它的函数或对象的结束中正确地超出作用域，确保智能指针拥有的对象被正确地释放。

c++ 11标准库附带4个智能指针类 : std::auto_ptr(在c++ 17中删除)，std::unique_ptr, std::shared_ptr和std::weak_ptr。unique_ptr是目前使用最多的智能指针类，因此我们将首先介绍它。在接下来的课程中，我们将讨论std::shared_ptr和std::weak_ptr。

---

std::unique_ptr

std::unique_ptr是c++ 11中对std::auto_ptr的替换。它应该被用于管理任何不由多个对象共享的动态分配对象。也就是说，std::unique_ptr应该完全拥有它所管理的对象，而不是与其他类共享这种所有权。std::unique_ptr存在于头文件<memory>中。

让我们看一个简单的智能指针的例子:

```
#include <iostream>
#include <memory> // for std::unique_ptr

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

int main()
{
  // allocate a Resource object and have it owned by std::unique_ptr
  std::unique_ptr<Resource> res{ new Resource() };

  return 0;
} // res goes out of scope here, and the allocated Resource is destroyed
```

因为上边的std::unique_ptr是在栈上分配的，所以它最终肯定会超出作用域，当它超出作用域时，它将删除它所管理的Resource。

与std::auto_ptr不同，std::unique_ptr正确实现了移动语义。

```
#include <iostream>
#include <memory> // for std::unique_ptr
```

```
#include <utility> // for std::move

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

int main()
{
  std::unique_ptr<Resource> res1{ new Resource{} }; // Resource created here
  std::unique_ptr<Resource> res2{}; // Start as nullptr

  std::cout << "res1 is " << (res1 ? "not null\n" : "null\n");
  std::cout << "res2 is " << (res2 ? "not null\n" : "null\n");

  // res2 = res1; // Won't compile: copy assignment is disabled
  res2 = std::move(res1); // res2 assumes ownership, res1 is set to null

  std::cout << "Ownership transferred\n";

  std::cout << "res1 is " << (res1 ? "not null\n" : "null\n");
  std::cout << "res2 is " << (res2 ? "not null\n" : "null\n");

  return 0;
} // Resource destroyed here when res2 goes out of scope
```

上述程序打印如下：

```
Resource acquired
res1 is not null
res2 is null
Ownership transferred
res1 is null
res2 is not null
Resource destroyed
```

因为std::unique_ptr在设计时考虑到了移动语义，所以复制构造和复制赋值是禁用的。如果要传输std::unique_ptr管理的内容，必须使用移动语义。在上面的程序中，我们通过std::move(将res1转换为r值，触发移动赋值而不是复制赋值)来实现这一点。

访问管理对象

unique_ptr重载了运算符operator*和operator→，可用于返回被管理的资源。operator*返回对托管资源的引用，operator→返回一个指针。

记住，std::unique_ptr可能并不总是在管理一个对象——要么是因为它是空创建的(使用默认构造函数或传入一个nullptr作为参数)，要么是因为它所管理的资源被移动到另一个std::unique_ptr。因此，在使用这两个操作符之前，应该检查std::unique_ptr是否实际上有资源。幸运的是，这很容易 : std::unique_ptr有一个转换为bool的类型，如果std::unique_ptr正在管理一个资源，则返回true。

这里有一个例子:

```
#include <iostream>
#include <memory> // for std::unique_ptr

class Resource
{
public:
  Resource() { std::cout << "Resource acquired\n"; }
  ~Resource() { std::cout << "Resource destroyed\n"; }
  friend std::ostream& operator<<(std::ostream& out, const Resource &res)
  {
    out << "I am a resource";
    return out;
  }
};

int main()
{
  std::unique_ptr<Resource> res{ new Resource{} };

  if (res) // use implicit cast to bool to ensure res contains a Resource
    std::cout << *res << '\n'; // print the Resource that res is owning

  return 0;
}
```

上述程序打印：

```
Resource acquired
I am a resource
Resource destroyed
```

在上面的程序中，我们使用重载操作符*来获取std::unique_ptr res拥有的Resource对象，然后将其发送给std::cout进行打印。

---

std::unique_ptr和数组

与std::auto_ptr不同，std::unique_ptr非常智能，可以知道是使用标量删除还是数组删除，因此std::unique_ptr可以同时用于标量对象和数组。

然而，与使用固定数组、动态数组或c风格字符串的std::unique_ptr相比，std::array或std::vector(或std::string)几乎总是更好的选择。

> 🙂 **Best practice**
> **优先使用std::array、std::vector或std::string，而不是管理固定数组、动态数组或c风格字符串的智能指针。**

std::make_unique

c++ 14附带了一个额外的函数std::make_unique()。这个模板函数构造一个模板类型的对象，并用传入函数的实参初始化它。

```
#include <memory> // for std::unique_ptr and std::make_unique
#include <iostream>

class Fraction
{
private:
  int m_numerator{ 0 };
  int m_denominator{ 1 };

public:
  Fraction(int numerator = 0, int denominator = 1) :
    m_numerator{ numerator }, m_denominator{ denominator }
  {
  }

  friend std::ostream& operator<<(std::ostream& out, const Fraction &f1)
  {
    out << f1.m_numerator << '/' << f1.m_denominator;
    return out;
  }
};


int main()
{
  // Create a single dynamically allocated Fraction with numerator 3 and denominator 5
  // We can also use automatic type deduction to good effect here
  auto f1{ std::make_unique<Fraction>(3, 5) };
  std::cout << *f1 << '\n';

  // Create a dynamically allocated array of Fractions of length 4
  auto f2{ std::make_unique<Fraction[]>(4) };
  std::cout << f2[0] << '\n';
```

```
  return 0;
}
```

上述程序打印：

```
3/5
0/1
```

使用std::make_unique()是可选的，但建议在自己创建std::unique_ptr之前使用。这是因为使用std::make_unique的代码更简单，而且它还需要更少的输入(当与自动类型推断一起使用时)。此外，它还解决了由于C++未指定函数参数的求值顺序而导致的**异常安全问题**。

> 🙂 **Best practice**
> **使用std::make_unique()，而不是使用std::unique_ptr和new来创建。**

异常安全问题的更多细节

对于那些想知道上面提到的“异常安全问题”是什么，这里有一个问题的描述。

考虑这样一个表达式:

```
some_function(std::unique_ptr<T>(new T), function_that_can_throw_exception());
```

编译器在如何处理这个调用方面有很大的灵活性。它可以创建一个新的T，然后调用function_that_can_throw_exception()，然后创建std::unique_ptr来管理动态分配的T。如果function_that_can_throw_exception()抛出一个异常，那么所分配的T将不会被释放，因为执行释放的智能指针还没有创建。这将导致T被泄露。

std::make_unique()不会遇到这个问题，因为对象T的创建和std::unique_ptr的创建都发生在std::make_unique()函数内部，在这里执行顺序没有歧义。

从函数返回std::unique_ptr

Std::unique_ptr可以安全地从函数中通过值返回:

```
#include <memory> // for std::unique_ptr

std::unique_ptr<Resource> createResource()
{
     return std::make_unique<Resource>();
}

int main()
{
    auto ptr{ createResource() };

    // do whatever

    return 0;
}
```

在上面的代码中，createResource()按值返回std::unique_ptr。如果这个值没有分配给任何东西，那么临时返回值将超出作用域，Resource将被清除。如果它被赋值(如main()所示)，在c++ 14或更早版本中，将使用移动语义将Resource从返回值转移到赋值给的对象(在上面的例子中，ptr)，而在c++ 17或更新版本中，将省略返回值。这使得通过std::unique_ptr返回资源比返回原始指针要安全得多!

一般来说，你不应该通过指针(永远)或引用返回std::unique_ptr(除非你有特定的令人信服的理由)。

将std::unique_ptr传递给函数

如果您希望函数拥有指针内容的所有权，请按值传递std::unique_ptr。注意，因为复制语义被禁用了，所以需要使用std::move来实际传递变量。

```
#include <iostream>
#include <memory> // for std::unique_ptr
#include <utility> // for std::move

class Resource
{
public:
  Resource() { std::cout << "Resource acquired\n"; }
  ~Resource() { std::cout << "Resource destroyed\n"; }
  friend std::ostream& operator<<(std::ostream& out, const Resource &res)
  {
    out << "I am a resource";
    return out;
  }
};

void takeOwnership(std::unique_ptr<Resource> res)
```

```
{
     if (res)
          std::cout << *res << '\n';
} // the Resource is destroyed here

int main()
{
    auto ptr{ std::make_unique<Resource>() };

//    takeOwnership(ptr); // This doesn't work, need to use move semantics
    takeOwnership(std::move(ptr)); // ok: use move semantics

    std::cout << "Ending program\n";

    return 0;
}
```

上述程序打印：

```
Resource acquired
I am a resource
Resource destroyed
Ending program
```

注意，在本例中，资源的所有权被转移到takeOwnership()，因此资源在takeOwnership()末尾而不是main()末尾被销毁。

但是，大多数情况下，您不希望函数拥有资源的所有权。尽管您可以通过引用传递std::unique_ptr(这将允许函数使用对象而不假定所有权)，但您应该只在被调用的函数可能更改被管理的对象时才这样做。

相反，最好是直接传递资源本身(通过指针或引用，取决于null是否是有效参数)。这允许函数不用了解调用者如何管理其资源。要从std::unique_ptr中获取原始资源指针，可以使用get()成员函数:

```
#include <memory> // for std::unique_ptr
#include <iostream>

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

  friend std::ostream& operator<<(std::ostream& out, const Resource &res)
  {
    out << "I am a resource";
    return out;
  }
};
```

```
// The function only uses the resource, so we'll accept a pointer to the resource, not
a reference to the whole std::unique_ptr<Resource>
void useResource(Resource* res)
{
  if (res)
    std::cout << *res << '\n';
  else
    std::cout << "No resource\n";
}

int main()
{
  auto ptr{ std::make_unique<Resource>() };

  useResource(ptr.get()); // note: get() used here to get a pointer to the Resource

  std::cout << "Ending program\n";

  return 0;
} // The Resource is destroyed here
```

上述程序打印：

```
Resource acquired
I am a resource
Ending program
Resource destroyed
```

---

std::unique_ptr和类

当然，您可以使用std::unique_ptr作为类的合成成员（composition member）。通过这种方式，您不必担心是否要确保类析构函数删除动态内存，因为当类对象被销毁时，std::unique_ptr将自动销毁。

然而，如果类对象没有被正确地销毁(例如，它是动态分配的，没有被正确地释放)，那么std::unique_ptr成员也不会被销毁，由std::unique_ptr管理的对象也不会被释放。

滥用std::unique_ptr

有两种容易误用std::unique_ptrs的方法，这两种方法都很容易避免。首先，不要让多个类管理相同的资源。例如:

```
Resource* res{ new Resource() };
std::unique_ptr<Resource> res1{ res };
std::unique_ptr<Resource> res2{ res };
```

虽然这在语法上是合法的，但最终结果将是res1和res2都试图删除Resource，这将导致未定义的行为。

其次，不要手动从std::unique_ptr下面删除资源。

```
Resource* res{ new Resource() };
std::unique_ptr<Resource> res1{ res };
delete res;
```

如果这样做，std::unique_ptr将尝试删除已经删除的资源，同样会导致未定义的行为。

注意std::make_unique()能防止上述两种情况的发生。