1、移动语义和智能指针的介绍

Chapter	M
Name	xingzp

考虑一个动态分配内存的函数:

```
void someFunction()
{
    Resource* ptr = new Resource(); // Resource is a struct or class

    // do stuff with ptr here

    delete ptr;
}
```

尽管上面的代码看起来相当简单，但是很容易忘记释放ptr。即使您记得在函数结束时删除ptr，如果函数提前退出，也有许多方法可以不删除ptr。这可以通过提前返回来实现:

```
#include <iostream>

void someFunction()
{
    Resource* ptr = new Resource();

    int x;
    std::cout << "Enter an integer: ";
```

```
    std::cin >> x;

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

    // do stuff with ptr here

    delete ptr;
}
```

或者通过抛出异常:

```
#include <iostream>

void someFunction()
{
    Resource* 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;
}
```

在上述两个程序中，早期的return或throw语句会执行，导致函数在没有删除变量ptr的情况下终止。因此，分配给变量ptr的内存现在被泄漏(并且将在每次调用此函数并提前返回时再次被泄漏)。

本质上，发生这类问题是因为指针变量没有内在的机制来进行自我清理。

智能指针能解决该问题?

类最好的一点是它们包含析构函数，当类的对象超出作用域时，析构函数会自动执行。因此，如果在构造函数中分配(或获取)内存，就可以在析构函数中释放它，并保证在类对象被销毁时释放内存(不管它是否超出作用域、显式删除等等)。这是我们在第13.9课—析构函数中讨论的RAII编程范式的核心。

那么，我们能否使用类来帮助我们管理和清理指针呢?我们可以!

考虑一个类，它的唯一工作是持有和“拥有”传递给它的指针，然后在类对象超出作用域时释放该指针。只要该类的对象只是作为局部变量创建的，我们就可以保证该类将正确地超出作用域(无论函数何时或如何终止)，并且拥有的指针将被销毁。

这是这个想法的初稿:

```
#include <iostream>

template <typename T>
class Auto_ptr1
{
  T* m_ptr;
public:
  // Pass in a pointer to "own" via the constructor
  Auto_ptr1(T* ptr=nullptr)
    :m_ptr(ptr)
  {
  }

  // The destructor will make sure it gets deallocated
  ~Auto_ptr1()
  {
    delete m_ptr;
  }

  // Overload dereference and operator-> so we can use Auto_ptr1 like m_ptr.
  T& operator*() const { return *m_ptr; }
  T* operator->() const { return m_ptr; }
};

// A sample class to prove the above works
class Resource
{
public:
    Resource() { std::cout << "Resource acquired\n"; }
    ~Resource() { std::cout << "Resource destroyed\n"; }
};

int main()
{
  Auto_ptr1<Resource> res(new Resource()); // Note the allocation of memory here

        // ... but no explicit delete needed

  // Also note that the Resource in angled braces doesn't need a * symbol, since tha
t's supplied by the template

  return 0;
} // res goes out of scope here, and destroys the allocated Resource for us
```

程序打印如下：

```
Resource acquired
Resource destroyed
```

考虑一下这个程序和类是如何工作的。首先，动态创建Resource，并将其作为参数传递给模板化的Auto_ptr1类。从这一点开始，Auto_ptr1变量res拥有该资源对象(Auto_ptr1与m_ptr有复合关系)。因为res被声明为局部变量并具有块作用域，所以当块结束时，它将超出作用域并被销毁(不用担心忘记释放它)。由于它是一个类，因此在销毁它时，将调用Auto_ptr1析构函数。该析构函数将确保它所持有的Resource指针被删除!

只要Auto_ptr1被定义为一个局部变量(具有自动生命周期，因此类名中的“Auto”部分)，无论函数如何终止(即使它提前终止)，Resource将保证在声明它的块的末尾被销毁。

这样的类称为智能指针。智能指针是一个组合类，设计用于动态管理分配的内存，并确保在智能指针对象超出作用域时删除内存。(与此相关，内置指针有时被称为“哑指针”，因为它们不能在自己之后进行清理)。

现在让我们回到上面的someFunction()例子，并展示智能指针类如何解决我们的挑战:

```
#include <iostream>

template <typename T>
class Auto_ptr1
{
  T* m_ptr;
public:
  // Pass in a pointer to "own" via the constructor
  Auto_ptr1(T* ptr=nullptr)
    :m_ptr(ptr)
  {
  }

  // The destructor will make sure it gets deallocated
  ~Auto_ptr1()
  {
    delete m_ptr;
  }

  // Overload dereference and operator-> so we can use Auto_ptr1 like m_ptr.
  T& operator*() const { return *m_ptr; }
  T* operator->() const { return m_ptr; }
};

// A sample class to prove the above works
class Resource
{
public:
    Resource() { std::cout << "Resource acquired\n"; }
```

```
    ~Resource() { std::cout << "Resource destroyed\n"; }
    void sayHi() { std::cout << "Hi!\n"; }
};

void someFunction()
{
    Auto_ptr1<Resource> ptr(new Resource()); // ptr now owns the Resource

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

    if (x == 0)
        return; // the function returns early

    // do stuff with ptr here
    ptr->sayHi();
}

int main()
{
    someFunction();

    return 0;
}
```

如果用户输入一个非零整数，上面的程序将打印:

```
Resource acquired
Hi!
Resource destroyed
```

如果用户输入0，上述程序将提前终止，打印:

```
Resource acquired
Resource destroyed
```

注意，即使在用户输入零且函数提前终止的情况下，资源仍然被正确地释放。

因为ptr变量是一个局部变量，所以ptr将在函数终止时被销毁(不管它以何种方式终止)。因为Auto_ptr1析构函数将清理Resource，所以我们可以保证Resource将被正确地清理。

一个严重的缺陷！

Auto_ptr1类在一些自动生成的代码背后隐藏着一个关键缺陷。在继续阅读之前，看看你是否能认出它是什么。我们会等待……

(提示:如果你不提供类的哪些部分会自动生成)

好了,时间到了。

我们不会告诉你，而是展示给你看。考虑下面的程序:

```
#include <iostream>

// Same as above
template <typename T>
class Auto_ptr1
{
  T* m_ptr;
public:
  Auto_ptr1(T* ptr=nullptr)
    :m_ptr(ptr)
  {
  }

  ~Auto_ptr1()
  {
    delete m_ptr;
  }

  T& operator*() const { return *m_ptr; }
  T* operator->() const { return m_ptr; }
};

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

int main()
{
  Auto_ptr1<Resource> res1(new Resource());
  Auto_ptr1<Resource> res2(res1); // Alternatively, don't initialize res2 and then ass
ign res2 = res1;

  return 0;
}
```

上述程序输出如下：

```
Resource acquired
Resource destroyed
Resource destroyed
```

您的程序很可能(但不一定)在此时崩溃。现在看到问题了吗?因为我们没有提供复制构造函数或赋值操作符，所以c++为我们提供了一个。它提供的函数只做浅拷贝。因此，当我们用res1初始化res2时，两个Auto_ptr1变量都指向同一个Resource。当res2超出作用域时，它删除资源，给res1留下一个悬空指针。当res1去删除它的(已经删除的)资源时，崩溃!

对于下述函数，你会遇到类似的问题:

```
void passByValue(Auto_ptr1<Resource> res)
{
}

int main()
{
  Auto_ptr1<Resource> res1(new Resource());
  passByValue(res1);

  return 0;
}
```

在这个程序中，res1将按值复制到passByValue的参数res中，导致Resource指针的复制。崩溃!

显然这不是什么好事。我们如何解决这个问题?

我们可以做的一件事是显式地定义和删除复制构造函数和赋值操作符，从而在第一时间阻止任何复制。这将防止按值传递的情况(这很好，我们可能不应该按值传递它们)。

但是，还有一个问题：我们如何从函数返回Auto_ptr1给调用者呢?

```
??? generateResource()
{
     Resource* r{ new Resource() };
     return Auto_ptr1(r);
}
```

我们不能通过引用返回Auto_ptr1，因为本地Auto_ptr1将在函数结束时被销毁，而调用者将剩下一个悬空引用。或者我们可以将指针r作为Resource*返回，但之后可能会忘记删除r（产生内存泄漏），这是使用智能指针的首要目的。所以这个想法也出局。按值

返回Auto_ptr1是唯一有意义的选项——但我们最终会得到浅拷贝、重复的指针和崩溃。

另一种选择是重写复制构造函数和赋值操作符来进行深度复制。这样，我们至少可以保证避免指向同一个对象的重复指针。但是复制可能会很昂贵(而且可能不需要甚至不可能)，而且我们不希望仅仅为了从函数返回Auto_ptr1而对对象进行不必要的复制。

我们该怎么办?

移动语义

如果不让复制构造函数和赋值操作符复制指针(“复制语义”)，而是将指针的所有权从源对象转移到目标对象，会怎样?这是move语义背后的核心思想。Move语义意味着类将转移对象的所有权，而不是进行复制。

让我们更新Auto_ptr1类来展示如何实现这一点:

```
#include <iostream>

template <typename T>
class Auto_ptr2
{
  T* m_ptr;
public:
  Auto_ptr2(T* ptr=nullptr)
    :m_ptr(ptr)
  {
  }

  ~Auto_ptr2()
  {
    delete m_ptr;
  }

  // A copy constructor that implements move semantics
  Auto_ptr2(Auto_ptr2& a) // note: not const
  {
    m_ptr = a.m_ptr; // transfer our dumb pointer from the source to our local object
    a.m_ptr = nullptr; // make sure the source no longer owns the pointer
  }

  // An assignment operator that implements move semantics
  Auto_ptr2& operator=(Auto_ptr2& a) // note: not const
  {
    if (&a == this)
      return *this;

    delete m_ptr; // make sure we deallocate any pointer the destination is already ho
lding first
    m_ptr = a.m_ptr; // then transfer our dumb pointer from the source to the local ob
```

```
ject
    a.m_ptr = nullptr; // make sure the source no longer owns the pointer
    return *this;
  }

  T& operator*() const { return *m_ptr; }
  T* operator->() const { return m_ptr; }
  bool isNull() const { return m_ptr == nullptr; }
};

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

int main()
{
  Auto_ptr2<Resource> res1(new Resource());
  Auto_ptr2<Resource> res2; // Start as nullptr

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

  res2 = res1; // res2 assumes ownership, res1 is set to null

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

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

  return 0;
}
```

上述程序打印：

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

注意，我们的重载的 **operator=** 将m_ptr的所有权从res1授予res2!因此，我们最终不会得到指针的重复副本，一切都得到了整洁的清理。

---

std::auto_ptr，为什么它是一个不好的尝试

现在是讨论std::auto_ptr的合适时机。std::auto_ptr在c++ 98中引入，在c++ 17中被删除，它是c++第一次尝试标准化智能指针。auto_ptr选择像Auto_ptr2类一样实现move语义。

然而，std::auto_ptr(以及我们的Auto_ptr2类)有许多问题，使得使用它很危险。

- 首先，由于std::auto_ptr通过复制构造函数和赋值操作符实现了移动语义，将std::auto_ptr按值传递给函数将导致您的资源被移动到函数形参中(并在函数形参超出作用域时销毁)。然后，当您从调用者上下文访问auto_ptr参数(没有意识到它被传输和删除了)时，您突然对空指针进行了解引用。崩溃!
- 其次，std::auto_ptr总是使用非数组删除来删除其内容。这意味着auto_ptr不能正确地处理动态分配的数组，因为它使用了错误的释放类型。更糟糕的是，它不能阻止您向它传递动态数组，这样它就会管理不当，导致内存泄漏。
- 最后，auto_ptr不能很好地处理标准库中的许多其他类，包括大多数容器和算法。这是因为那些标准库类假定当它们复制一个项时，它实际上进行了复制，而不是移动。

由于上述缺点，std::auto_ptr在c++ 11中已弃用，在c++ 17中已删除。

接下来

std::auto_ptr设计的核心问题是，在c++ 11之前，c++语言根本没有区分“复制语义”和“移动语义”的机制。覆盖复制语义来实现移动语义会导致奇怪的边缘情况和无心的错误。例如，你可以写res1 = res2，但不知道res2是否会被改变!

正因为如此，在c++ 11中，正式定义了“移动”的概念，并在语言中添加了“移动语义”，以正确区分复制和移动。既然我们已经为为什么移动语义是有用的奠定了基础，我们将在本章的其余部分探索移动语义的主题。我们还将使用“移动语义”修复Auto_ptr2类。

在c++ 11中，std::auto_ptr已经被一堆其他类型的“移动感知”智能指针所取代:std::unique_ptr, std::weak_ptr和std::shared_ptr。我们还将探讨其中最流行的两个:unique_ptr(它直接替换了auto_ptr)和shared_ptr。