M.1 — Intro to smart pointers and move semantics

ALEX JANUARY 25, 2022

Consider a function in which we dynamically allocate a value:

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

    // do stuff with ptr here

    delete ptr;
}
```

Although the above code seems fairly straightforward, it's fairly easy to forget to deallocate ptr. Even if you do remember to delete ptr at the end of the function, there are a myriad of ways that ptr may not be deleted if the function exits early. This can happen via an early return:

```
#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;
}
```

or via a thrown exception:

```
#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;
}
```

In the above two programs, the early return or throw statement execute, causing the function to terminate without variable ptr being deleted. Consequently, the memory allocated for variable ptr is now leaked (and will be leaked again every time this function is called and returns early).

At heart, these kinds of issues occur because pointer variables have no inherent mechanism to clean up after themselves.

Smart pointer classes to the rescue?

One of the best things about classes is that they contain destructors that automatically get executed when an object of the class goes out of scope. So if you allocate (or acquire) memory in your constructor, you can deallocate it in your destructor, and be guaranteed that the memory will be deallocated when the class object is destroyed (regardless of whether it goes out of scope, gets explicitly deleted, etc…). This is at the heart of the RAII programming paradigm that we talked about in lesson 12.9 -- Destructors.

So can we use a class to help us manage and clean up our pointers? We can!

Consider a class whose sole job was to hold and "own" a pointer passed to it, and then deallocate that pointer when the class object went out of scope. As long as objects of that class were only created as local variables, we could guarantee that the class would properly go out of scope (regardless of when or how our functions terminate) and the owned pointer would get destroyed.

Here's a first draft of the idea:

```
#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 用这个类去实例化上面的类模板
{
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 that's supplied by the template

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

This program prints:

```
Resource acquired
Resource destroyed
```

Consider how this program and class work. First, we dynamically create a Resource, and pass it as a parameter to our templated Auto_ptr1 class. From that point forward, our Auto_ptr1 variable res owns that Resource object (Auto_ptr1 has a composition relationship with m_ptr). Because res is declared as a local variable and has block scope, it will go out of scope when the block ends, and be destroyed (no worries about forgetting to deallocate it). And because it is a class, when it is destroyed, the Auto_ptr1 destructor will be called. That destructor will ensure that the Resource pointer it is holding gets deleted!

As long as Auto_ptr1 is defined as a local variable (with automatic duration, hence the "Auto" part of the class name), the Resource will be guaranteed to be destroyed at the end of the block it is declared in, regardless of how the function terminates (even if it terminates early).

Such a class is called a smart pointer. A **Smart pointer** is a composition class that is designed to manage dynamically allocated memory and ensure that memory gets deleted when the smart pointer object goes out of scope. (Relatedly, built-in pointers are sometimes called "dumb pointers" because they can't clean up after themselves).

Now let's go back to our someFunction() example above, and show how a smart pointer class can solve our challenge:

```
#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;
}
```

If the user enters a non-zero integer, the above program will print:

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

If the user enters zero, the above program will terminate early, printing:

```
Resource acquired
Resource destroyed
```

Note that even in the case where the user enters zero and the function terminates early, the Resource is still properly deallocated.

Because the ptr variable is a local variable, ptr will be destroyed when the function terminates (regardless of how it terminates). And because the Auto_ptr1 destructor will clean up the Resource, we are assured that the Resource will be properly cleaned up.

A critical flaw

lurk [lɜːk]
vi. 潜伏；潜藏；埋伏
n. 潜伏；埋伏

The Auto_ptr1 class has a critical flaw lurking behind some auto-generated code. Before reading further, see if you can identify what it is. We'll wait…

(Hint: consider what parts of a class get auto-generated if you don't supply them)

(Jeopardy music)

Okay, time's up.

Rather than tell you, we'll show you. Consider the following program:

```
#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 assign res2 =
res1;

    return 0;
}
```

默认的 [赋值运算符] 和 [拷贝构造函数] 都是浅拷贝, 故指针的赋值运算和拷贝构造要注意浅拷贝
int x = 1 ;
int * p1 = &x;
int * p2
① p2 = p1 调用[赋值运算符], 指针赋给指针
② p2(p1) 调用[拷贝构造函数], 指针拷贝指针

注意:这里的对象不是指针, 而是含有指针成员的对象.

综上,

变量清理的时候是倒着来清理的, res2先被清理, res1再被清理

This program prints:

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

Very likely (but not necessarily) your program will crash at this point. See the problem now? Because we haven't supplied a copy constructor or an assignment operator, C++ provides one for us. And the functions it provides do shallow copies. So when we initialize res2 with res1, both Auto_ptr1 variables are pointed at the same Resource. When res2 goes out of the scope, it deletes the resource, leaving res1 with a dangling pointer. When res1 goes to delete its (already deleted) Resource, crash!

You'd run into a similar problem with a function like this:

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

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

    return 0;
}
```

形参res和实参res1都是含有指针成员的对象
把res1按值传递,就是把res1的各个成员,拷贝给res的各个成员.
这时两者的指针成员之间的拷贝就会按照默认的[复制构造函数]做浅拷贝.

多个指针指向同一个东东, 一个指针清理后, 别的指针就成游荡指针了

In this program, res1 will be copied by value into passByValue's parameter res, leading to duplication of the Resource pointer. Crash!

So clearly this isn't good. How can we address this?

Well, one thing we could do would be to explicitly define and delete the copy constructor and assignment operator, thereby preventing any copies from being made in the first place. That would prevent the pass by value case (which is good, we probably shouldn't be passing these by value anyway).

But then how would we return an Auto_ptr1 from a function back to the caller?

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

We can't return our Auto_ptr1 by reference, because the local Auto_ptr1 will be destroyed at the end of the function, and the caller will be left with a dangling reference. We could return pointer r as `Resource*`, but then we might forget to delete r later, which is the whole point of using smart pointers in the first place. So that's out. Returning the Auto_ptr1 by value is the only option that makes sense -- but then we end up with shallow copies, duplicated pointers, and crashes.

Another option would be to override the copy constructor and assignment operator to make deep copies. In this way, we'd at least guarantee to avoid duplicate pointers to the same object. But copying can be expensive (and may not be desirable or even possible), and we don't want to make needless copies of objects just to return an Auto_ptr1 from a function. Plus assigning or initializing a dumb pointer doesn't copy the object being pointed to, so why would we expect smart pointers to behave differently?

What do we do?

Move semantics

What if, instead of having our copy constructor and assignment operator copy the pointer ("copy semantics"), we instead transfer/move ownership of the pointer from the source to the destination object? This is the core idea behind move semantics. **Move semantics** means the class will transfer ownership of the object rather than making a copy.

Let's update our Auto_ptr1 class to show how this can be done:

```
#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 holding first
        m_ptr = a.m_ptr; // then transfer our dumb pointer from the source to the local object
        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;
}
```

Atuo_ptr2&:
Auto_ptr2对象的引用,而不是Atuo_ptr2指针的引用
Auto_ptr2对象内本身含有指针

This program prints:

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

Note that our overloaded operator= gave ownership of m_ptr from res1 to res2! Consequently, we don't end up with duplicate copies of the pointer, and everything gets tidily cleaned up.

std::auto_ptr, and why it was a bad idea

Now would be an appropriate time to talk about std::auto_ptr. std::auto_ptr, introduced in C++98 and removed in C++17, was C++'s first attempt at a standardized smart pointer. std::auto_ptr opted to implement move semantics just like the Auto_ptr2 class does.

However, std::auto_ptr (and our Auto_ptr2 class) has a number of problems that makes using it dangerous.

First, because std::auto_ptr implements move semantics through the copy constructor and assignment operator, passing a std::auto_ptr by value to a function will cause your resource to get moved to the function parameter (and be destroyed at the end of the function when the function parameters go out of scope). Then when you go to access your auto_ptr argument from the caller (not realizing it was transferred and deleted), you're suddenly dereferencing a null pointer. Crash!

Second, std::auto_ptr always deletes its contents using non-array delete. This means auto_ptr won't work correctly with dynamically allocated arrays, because it uses the wrong kind of deallocation. Worse, it won't prevent you from passing it a dynamic array, which it will then mismanage, leading to memory leaks.

Finally, auto_ptr doesn't play nice with a lot of the other classes in the standard library, including most of the containers and algorithms. This occurs because those standard library classes assume that when they copy an item, it actually makes a copy, not a move.

Because of the above mentioned shortcomings, std::auto_ptr has been deprecated in C++11 and removed in C++17.

Moving forward

The core problem with the design of std::auto_ptr is that prior to C++11, the C++ language simply had no mechanism to differentiate "copy semantics" from "move semantics". Overriding the copy semantics to implement move semantics leads to weird edge cases and inadvertent bugs. For example, you can write `res1 = res2` and have no idea whether res2 will be changed or not!

Because of this, in C++11, the concept of "move" was formally defined, and "move semantics" were added to the language to properly differentiate copying from moving. Now that we've set the stage for why move semantics can be useful, we'll explore the topic of move semantics throughout the rest of this chapter. We'll also fix our Auto_ptr2 class using move semantics.

In C++11, std::auto_ptr has been replaced by a bunch of other types of "move-aware" smart pointers: std::unique_ptr, std::weak_ptr, and std::shared_ptr. We'll also explore the two most popular of these: unique_ptr (which is a direct replacement for auto_ptr) and shared_ptr.

Next lesson
M.2 R-value references

Back to table of contents

Previous lesson
20.x Chapter 20 comprehensive quiz