

## 02 | 自己动手，实现C++的智能指针

2019-11-25 吴咏炜

现代C++实战30讲

[进入课程 >](#)



讲述：吴咏炜

时长 14:11 大小 9.75M



你好，我是吴咏炜。

上一讲，我们描述了一个某种程度上可以当成智能指针用的类 `shape_wrapper`。使用那个智能指针，可以简化资源的管理，从根本上消除资源（包括内存）泄漏的可能性。这一讲我们就来进一步讲解，如何将 `shape_wrapper` 改造成一个完整的智能指针。你会看到，智能指针本质上并不神秘，其实就是 RAII 资源管理功能的自然展现而已。

在学完这一讲之后，你应该会对 C++ 的 `unique_ptr` 和 `shared_ptr` 的功能非常熟悉了。同时，如果你今后要创建类似的资源管理类，也不会是一件难事。

回顾

我们上一讲给出了下面这个类：

 复制代码

```
1 class shape_wrapper {
2 public:
3     explicit shape_wrapper(
4         shape* ptr = nullptr)
5         : ptr_(ptr) {}
6     ~shape_wrapper()
7     {
8         delete ptr_;
9     }
10    shape* get() const { return ptr_; }
11
12 private:
13     shape* ptr_;
14 };
```

这个类可以完成智能指针的最基本的功能：对超出作用域的对象进行释放。但它缺了点东西：

1. 这个类只适用于 `shape` 类
2. 该类对象的行为不够像指针
3. 拷贝该类对象会引发程序行为异常

下面我们来逐一看一下怎么弥补这些问题。

## 模板化和易用性

要让这个类能够包装任意类型的指针，我们需要把它变成一个类模板。这实际上相当容易：

 复制代码

```
1 template <typename T>
2 class smart_ptr {
3 public:
4     explicit smart_ptr(T* ptr = nullptr)
5         : ptr_(ptr) {}
6     ~smart_ptr()
7     {
8         delete ptr_;
9     }
```

```
10     T* get() const { return ptr_; }
11 private:
12     T* ptr_;
13 };
```

和 `shape_wrapper` 比较一下，我们就是在开头增加模板声明 `template <typename T>`，然后把代码中的 `shape` 替换成模板参数 `T` 而已。这些修改非常简单自然吧？模板本质上并不是一个很复杂的概念。这个模板使用也很简单，把原来的 `shape_wrapper` 改成 `smart_ptr<shape>` 就行。

目前这个 `smart_ptr` 的行为还是和指针有点差异的：


它不能用 `*` 运算符解引用

它不能用 `->` 运算符指向对象成员

它不能像指针一样用在布尔表达式里

不过，这些问题也相当容易解决，加几个成员函数就可以：


```
1 template <typename T>
2 class smart_ptr {
3 public:
4     ...
5     T& operator*() const { return *ptr_; }
6     T* operator->() const { return ptr_; }
7     operator bool() const { return ptr_; }
8 }
```

 复制代码

## 拷贝构造和赋值


拷贝构造和赋值，我们暂且简称为拷贝，这是个比较复杂的问题了。关键还不是实现问题，而是我们该如何定义其行为。假设有下面的代码：

```
1 smart_ptr<shape> ptr1{create_shape(shape_type::circle)};
2 smart_ptr<shape> ptr2{ptr1};
```

 复制代码

对于第二行，究竟应当让编译时发生错误，还是可以有一个更合理的行为？我们来逐一检查一下各种可能性。

最简单的情况显然是禁止拷贝。我们可以使用下面的代码：

 复制代码

```
1  template <typename T>
2  class smart_ptr {
3      ...
4      smart_ptr(const smart_ptr&)
5          = delete;
6      smart_ptr& operator=(const smart_ptr&)
7          = delete;
8      ...
9  };
```

禁用这两个函数非常简单，但却解决了一种可能出错的情况。否则，`smart_ptr<shape> ptr2{ptr1};` 在编译时不会出错，但在运行时却会有未定义行为——由于会对同一内存释放两次，通常情况下会导致程序崩溃。

我们是不是可以考虑在拷贝智能指针时把对象拷贝一份？不行，通常人们不会这么用，因为使用智能指针的目的就是要减少对象的拷贝啊。何况，虽然我们的指针类型是 `shape`，但实际指向的却应该是 `circle` 或 `triangle` 之类的对象。在 C++ 里没有像 Java 的 `clone` 方法这样的约定；一般而言，并没有通用的方法可以通过基类的指针来构造出一个子类的对象来。

我们要么试试在拷贝时转移指针的所有权？大致实现如下：

 复制代码

```
1  template <typename T>
2  class smart_ptr {
3      ...
4      smart_ptr(smart_ptr& other)
5      {
6          ptr_ = other.release();
7      }
8      smart_ptr& operator=(smart_ptr& rhs)
9      {
```

```

10     smart_ptr(rhs).swap(*this);
11     return *this;
12 }
13 ...
14 T* release()
15 {
16     T* ptr = ptr_;
17     ptr_ = nullptr;
18     return ptr;
19 }
20 void swap(smart_ptr& rhs)
21 {
22     using std::swap;
23     swap(ptr_, rhs.ptr_);
24 }
25 ...
26 };

```

在拷贝构造函数中，通过调用 `other` 的 `release` 方法来释放它对指针的所有权。在赋值函数中，则通过拷贝构造产生一个临时对象并调用 `swap` 来交换对指针的所有权。实现上是不复杂的。

如果你学到的赋值函数还有一个类似于 `if (this != &rhs)` 的判断的话，那种用法更啰嗦，而且异常安全性不够好——如果在赋值过程中发生异常的话，`this` 对象的内容可能已经被部分破坏了，对象不再处于一个完整的状态。

**目前这种惯用法（见参考资料 [1]）则保证了强异常安全性：**赋值分为拷贝构造和交换两步，异常只可能在第一步发生；而第一步如果发生异常的话，`this` 对象完全不受任何影响。无论拷贝构造成功与否，结果只有赋值成功和赋值没有效果两种状态，而不会发生因为赋值破坏了当前对象这种场景。

如果你觉得这个实现还不错的话，那恭喜你，你达到了 C++ 委员会在 1998 年时的水平：上面给出的语义本质上就是 C++98 的 `auto_ptr` 的定义。如果你觉得这个实现很别扭的话，也恭喜你，因为 C++ 委员会也是这么觉得的：`auto_ptr` 在 C++17 时已经被正式从 C++ 标准里删除了。

上面实现的最大问题是，它的行为会让程序员非常容易犯错。一不小心把它传递给另外一个 `smart_ptr`，你就不再拥有这个对象了.....

## “移动”指针？

在下一讲我们将完整介绍一下移动语义。这一讲，我们先简单看一下 `smart_ptr` 可以如何使用“移动”来改善其行为。

我们需要对代码做两处小修改：

 复制代码

```
1  template <typename T>
2  class smart_ptr {
3      ...
4      smart_ptr(smart_ptr&& other)
5      {
6          ptr_ = other.release();
7      }
8      smart_ptr& operator=(smart_ptr rhs)
9      {
10         rhs.swap(*this);
11         return *this;
12     }
13     ...
14 };
```

看到修改的地方了吗？我改了两个地方：

把拷贝构造函数中的参数类型 `smart_ptr&` 改成了 `smart_ptr&&`；现在它成了移动构造函数。

把赋值函数中的参数类型 `smart_ptr&` 改成了 `smart_ptr`，在构造参数时直接生成新的智能指针，从而不再需要在函数体中构造临时对象。现在赋值函数的行为是移动还是拷贝，完全依赖于构造参数时走的是移动构造还是拷贝构造。

根据 C++ 的规则，如果我提供了移动构造函数而没有手动提供拷贝构造函数，那后者自动被禁用（记住，C++ 里那些复杂的规则也是为方便编程而设立的）。于是，我们自然地得到了以下结果：

 复制代码

```
1  smart_ptr<shape> ptr1{create_shape(shape_type::circle)};
2  smart_ptr<shape> ptr2{ptr1};           // 编译出错
3  smart_ptr<shape> ptr3;
```

```
4 ptr3 = ptr1; // 编译出错
5 ptr3 = std::move(ptr1); // OK, 可以
6 smart_ptr<shape> ptr4{std::move(ptr3)}; // OK, 可以
```

这个就自然多了。

这也是 C++11 的 `unique_ptr` 的基本行为。

## 子类指针向基类指针的转换

哦，我撒了一个小谎。不知道你注意到没有，一个 `circle*` 是可以隐式转换成 `shape*` 的，但上面的 `smart_ptr<circle>` 却无法自动转换成 `smart_ptr<shape>`。这个行为显然还是不够“自然”。

不过，只需要额外加一点模板代码，就能实现这一行为。在我们目前给出的实现里，只需要修改我们的移动构造函数一处即可——这也算是我们让赋值函数使用拷贝 / 移动构造函数的好处了。

```
1  template <typename U>
2  smart_ptr(smart_ptr<U>&& other)
3  {
4      ptr_ = other.release();
5  }
```

 复制代码

这样，我们自然而然利用了指针的转换特性：现在 `smart_ptr<circle>` 可以移动给 `smart_ptr<shape>`，但不能移动给 `smart_ptr<triangle>`。不正确的转换会在代码编译时直接报错。

至于非隐式的转换，因为本来就是要写特殊的转换函数的，我们留到这一讲的最后再讨论。

## 引用计数

`unique_ptr` 算是一种较为安全的智能指针了。但是，一个对象只能被单个 `unique_ptr` 所拥有，这显然不能满足所有使用场合的需求。一种常见的情况是，多个智能指针同时拥有



一个对象；当它们全部都失效时，这个对象也同时会被删除。这也就是 `shared_ptr` 了。

`unique_ptr` 和 `shared_ptr` 的主要区别如下图所示：

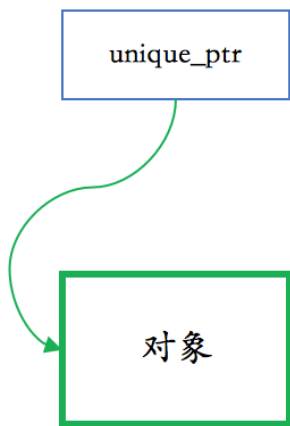


图1a: `unique_ptr`

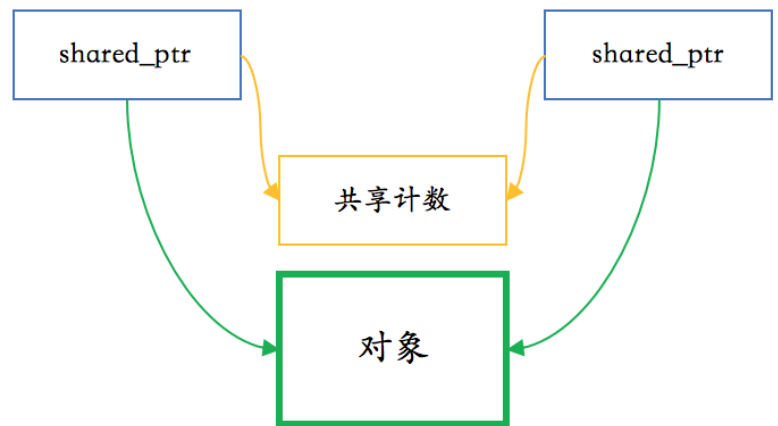


图1b: `shared_ptr`

多个不同的 `shared_ptr` 不仅可以共享一个对象，在共享同一对象时也需要同时共享同一个计数。当最后一个指向对象（和共享计数）的 `shared_ptr` 析构时，它需要删除对象和共享计数。我们下面就来实现一下。

我们先来写出共享计数的接口：

```
1 class shared_count {
2 public:
3     shared_count();
4     void add_count();
5     long reduce_count();
6     long get_count() const;
7 };
```

[复制代码](#)

这个 `shared_count` 类除构造函数之外有三个方法：一个增加计数，一个减少计数，一个获取计数。注意上面的接口增加计数不需要返回计数值；但减少计数时需要返回计数值，以供调用者判断是否它已经是最后一个指向共享计数的 `shared_ptr` 了。由于真正多线程安全的版本需要用到我们目前还没学到的知识，我们目前先实现一个简单化的版本：



```
1 class shared_count {
2 public:
3     shared_count() : count_(1) {}
4     void add_count()
5     {
6         ++count_;
7     }
8     long reduce_count()
9     {
10         return --count_;
11     }
12     long get_count() const
13     {
14         return count_;
15     }
16
17 private:
18     long count_;
19 };
```


现在我们可以实现我们的引用计数智能指针了。首先是构造函数、析构函数和私有成员变量：

```
1 template <typename T>
2 class smart_ptr {
3 public:
4     explicit smart_ptr(T* ptr = nullptr)
5         : ptr_(ptr)
6     {
7         if (ptr) {
8             shared_count_ =
9                 new shared_count();
10        }
11    }
12    ~smart_ptr()
13    {
14        if (ptr_ &&
15            !shared_count_
16            ->reduce_count()) {
17            delete ptr_;
18            delete shared_count_;
19        }
20    }
21
22 private:
```

```
23     T* ptr_;
24     shared_count* shared_count_;
25 };
```


构造函数跟之前的主要不同点是会构造一个 `shared_count` 出来。析构函数在看到 `ptr_` 非空时（此时根据代码逻辑，`shared_count` 也必然非空），需要对引用数减一，并在引用数降到零时彻底删除对象和共享计数。原理就是这样，不复杂。

当然，我们还有些细节要处理。为了方便实现赋值（及其他一些惯用法），我们需要一个新的 `swap` 成员函数：

 复制代码

```
1     void swap(smart_ptr& rhs)
2     {
3         using std::swap;
4         swap(ptr_, rhs.ptr_);
5         swap(shared_count_,
6              rhs.shared_count_);
7     }
```

赋值函数可以跟前面一样，保持不变，但拷贝构造和移动构造函数是需要更新一下的：

 复制代码

```
1     template <typename U>
2     smart_ptr(const smart_ptr<U>& other)
3     {
4         ptr_ = other.ptr_;
5         if (ptr_) {
6             other.shared_count_
7                 ->add_count();
8             shared_count_ =
9                 other.shared_count_;
10        }
11    }
12    template <typename U>
13    smart_ptr(smart_ptr<U>&& other)
14    {
15        ptr_ = other.ptr_;
16        if (ptr_) {
17            shared_count_ =
18                other.shared_count_;
19            other.ptr_ = nullptr;
```

```
20     }
21 }
```


除复制指针之外，对于拷贝构造的情况，我们需要在指针非空时把引用数加一，并复制共享计数的指针。对于移动构造的情况，我们不需要调整引用数，直接把 `other.ptr_` 置为空，认为 `other` 不再指向该共享对象即可。

不过，上面的代码有个问题：它不能正确编译。编译器会报错，像：

```
fatal error: 'ptr_' is a private member of 'smart_ptr<circle>'
```


错误原因是模板的各个实例间并不天然就有 `friend` 关系，因而不能互访私有成员 `ptr_` 和 `shared_count_`。我们需要在 `smart_ptr` 的定义中显式声明：

```
1  template <typename U>
2  friend class smart_ptr;
```

 复制代码

此外，我们之前的实现（类似于单一所有权的 `unique_ptr`）中用 `release` 来手工释放所有权。在目前的引用计数实现中，它就不太合适了，应当删除。但我们要加一个对调试非常有用的函数，返回引用计数值。定义如下：

```
1  long use_count() const
2  {
3      if (ptr_) {
4          return shared_count_
5              ->get_count();
6      } else {
7          return 0;
8      }
9  }
```

 复制代码

这就差不多是一个比较完整的引用计数智能指针的实现了。我们可以用下面的代码来验证一下它的功能正常：

```
1 class shape {
2 public:
3     virtual ~shape() {}
4 };
5
6 class circle : public shape {
7 public:
8     ~circle() { puts("~circle()"); }
9 };
10
11 int main()
12 {
13     smart_ptr<circle> ptr1(new circle());
14     printf("use count of ptr1 is %ld\n",
15           ptr1.use_count());
16     smart_ptr<shape> ptr2;
17     printf("use count of ptr2 was %ld\n",
18           ptr2.use_count());
19     ptr2 = ptr1;
20     printf("use count of ptr2 is now %ld\n",
21           ptr2.use_count());
22     if (ptr1) {
23         puts("ptr1 is not empty");
24     }
25 }
```

这段代码的运行结果是：

use count of ptr1 is 1

use count of ptr2 was 0

use count of ptr2 is now 2

ptr1 is not empty

~circle()

上面我们可以看到引用计数的变化，以及最后对象被成功删除。

## 指针类型转换

对应于 C++ 里的不同的类型强制转换：

`static_cast`

`reinterpret_cast`

`const_cast`

`dynamic_cast`

智能指针需要实现类似的函数模板。实现本身并不复杂，但为了实现这些转换，我们需要添加构造函数，允许在对智能指针内部的指针对象赋值时，使用一个现有的智能指针的共享计数。如下所示：

 复制代码

```
1  template <typename U>
2  smart_ptr(const smart_ptr<U>& other,
3            T* ptr)
4  {
5      ptr_ = ptr;
6      if (ptr_) {
7          other.shared_count_
8              ->add_count();
9          shared_count_ =
10             other.shared_count_;
11     }
12 }
```

这样我们就可以实现转换所需的函数模板了。下面实现一个 `dynamic_pointer_cast` 来示例一下：

 复制代码

```
1  template <typename T, typename U>
2  smart_ptr<T> dynamic_pointer_cast(
3      const smart_ptr<U>& other)
4  {
5      T* ptr =
6          dynamic_cast<T*>(other.get());
7      return smart_ptr<T>(other, ptr);
8  }
```

在前面的验证代码后面我们可以加上：

 复制代码

```
1 smart_ptr<circle> ptr3 =  
2     dynamic_pointer_cast<circle>(ptr2);  
3 printf("use count of ptr3 is %ld\n",  
4     ptr3.use_count());
```

编译会正常通过，同时能在输出里看到下面的结果：

```
use count of ptr3 is 3
```

最后，对象仍然能够被正确删除。这说明我们的实现是正确的。

## 代码列表

为了方便你参考，下面我给出了一个完整的 `smart_ptr` 代码列表：

 复制代码

```
1 #include <utility> // std::swap  
2  
3 class shared_count {  
4 public:  
5     shared_count() noexcept  
6         : count_(1) {}  
7     void add_count() noexcept  
8     {  
9         ++count_;  
10    }  
11    long reduce_count() noexcept  
12    {  
13        return --count_;  
14    }  
15    long get_count() const noexcept  
16    {  
17        return count_;  
18    }  
19  
20 private:  
21     long count_;  
22 };  
23  
24 template <typename T>
```

```

25 class smart_ptr {
26 public:
27     template <typename U>
28     friend class smart_ptr;
29
30     explicit smart_ptr(T* ptr = nullptr)
31         : ptr_(ptr)
32     {
33         if (ptr) {
34             shared_count_ =
35                 new shared_count();
36         }
37     }
38     ~smart_ptr()
39     {
40         printf("~smart_ptr(): %p\n", this);
41         if (ptr_ &&
42             !shared_count_
43             ->reduce_count()) {
44             delete ptr_;
45             delete shared_count_;
46         }
47     }
48
49     template <typename U>
50     smart_ptr(const smart_ptr<U>& other) noexcept
51     {
52         ptr_ = other.ptr_;
53         if (ptr_) {
54             other.shared_count_->add_count();
55             shared_count_ = other.shared_count_;
56         }
57     }
58     template <typename U>
59     smart_ptr(smart_ptr<U>&& other) noexcept
60     {
61         ptr_ = other.ptr_;
62         if (ptr_) {
63             shared_count_ =
64                 other.shared_count_;
65             other.ptr_ = nullptr;
66         }
67     }
68     template <typename U>
69     smart_ptr(const smart_ptr<U>& other,
70               T* ptr) noexcept
71     {
72         ptr_ = ptr;
73         if (ptr_) {
74             other.shared_count_
75                 ->add_count();
76             shared_count_ =

```



```

77         other.shared_count_;
78     }
79 }
80 smart_ptr&
81 operator=(smart_ptr rhs) noexcept
82 {
83     rhs.swap(*this);
84     return *this;
85 }
86
87 T* get() const noexcept
88 {
89     return ptr_;
90 }
91 long use_count() const noexcept
92 {
93     if (ptr_) {
94         return shared_count_
95             ->get_count();
96     } else {
97         return 0;
98     }
99 }
100 void swap(smart_ptr& rhs) noexcept
101 {
102     using std::swap;
103     swap(ptr_, rhs.ptr_);
104     swap(shared_count_,
105          rhs.shared_count_);
106 }
107
108 T& operator*() const noexcept
109 {
110     return *ptr_;
111 }
112 T* operator->() const noexcept
113 {
114     return ptr_;
115 }
116 operator bool() const noexcept
117 {
118     return ptr_;
119 }
120
121 private:
122     T* ptr_;
123     shared_count* shared_count_;
124 };
125
126 template <typename T>
127 void swap(smart_ptr<T>& lhs,
128          smart_ptr<T>& rhs) noexcept

```

```

129 {
130     lhs.swap(rhs);
131 }
132
133 template <typename T, typename U>
134 smart_ptr<T> static_pointer_cast(
135     const smart_ptr<U>& other) noexcept
136 {
137     T* ptr = static_cast<T*>(other.get());
138     return smart_ptr<T>(other, ptr);
139 }
140
141 template <typename T, typename U>
142 smart_ptr<T> reinterpret_pointer_cast(
143     const smart_ptr<U>& other) noexcept
144 {
145     T* ptr = reinterpret_cast<T*>(other.get());
146     return smart_ptr<T>(other, ptr);
147 }
148
149 template <typename T, typename U>
150 smart_ptr<T> const_pointer_cast(
151     const smart_ptr<U>& other) noexcept
152 {
153     T* ptr = const_cast<T*>(other.get());
154     return smart_ptr<T>(other, ptr);
155 }
156
157 template <typename T, typename U>
158 smart_ptr<T> dynamic_pointer_cast(
159     const smart_ptr<U>& other) noexcept
160 {
161     T* ptr = dynamic_cast<T*>(other.get());
162     return smart_ptr<T>(other, ptr);
163 }

```

如果你足够细心的话，你会发现我在代码里加了不少 `noexcept`。这对这个智能指针在它的目标场景能正确使用是十分必要的。我们会在下面的几讲里回到这个话题。

## 内容小结

这一讲我们从 `shape_wrapper` 出发，实现了一个基本完整的带引用计数的智能指针。这个智能指针跟标准的 `shared_ptr` 比，还缺了一些东西（见参考资料 [2]），但日常用到的智能指针功能已经包含在内。现在，你应当已经对智能指针有一个较为深入的理解了。

## 课后思考

这里留几个问题，你可以思考一下：

1. 不查阅 `shared_ptr` 的文档，你觉得目前 `smart_ptr` 应当添加什么功能吗？
2. 你想到的功能在标准的 `shared_ptr` 里吗？
3. 你觉得智能指针应该满足什么样的线程安全性？

欢迎留言和我交流你的看法。

## 参考资料

[1] Stack Overflow, GManNickG' s answer to “What is the copy-and-swap idiom?” . <https://stackoverflow.com/a/3279550/816999>

[2] cppreference.com, “std::shared\_ptr” .  
[https://en.cppreference.com/w/cpp/memory/shared\\_ptr](https://en.cppreference.com/w/cpp/memory/shared_ptr)


点击查看 

# 打卡学习 C++ 拒绝从入门到放弃



PC 端用户扫码参与



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

## 精选留言

写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。