

1、C++ 11有哪些新特性？

- nullptr替代 NULL
- 引入了 auto 和 decltype 这两个关键字实现了类型推导
- 基于范围的 for 循环for(auto& i : res){}
- 类和结构体的静态初始化列表
- 成员变量默认初始化
- 右值引用和move语义
- 智能指针等

标准库的扩充（往STL里新加进一些模板类，比较好用）

- Lambda 表达式（匿名函数）
- std::forward_list（单向链表）
- 无序容器和正则表达式

答案解析

(1)为什么要有decltype 因为 auto 并不适用于所有的自动类型推导场景，在某些特殊情况下 auto 用起来非常不方便，甚至压根无法使用，所以 decltype 关键字也被引入到 C++11 中。

auto 和 decltype 关键字都可以自动推导出变量的类型，但它们的用法是有区别的：

```
1 | auto varname = value;
2 | decltype(exp) varname = value;
```

其中，varname 表示变量名，value 表示赋给变量的值，exp 表示一个表达式。

auto 根据"="右边的初始值 value 推导出变量的类型，而 decltype 根据 exp 表达式推导出变量的类型，跟"="右边的 value 没有关系。

另外，auto 要求变量必须初始化，而 decltype 不要求。这很容易理解，auto 是根据变量的初始值来推导出变量类型的，如果不初始化，变量的类型也就无法推导了。decltype 可以写成下面的形式：

```
1 | decltype(exp) varname;
```

2、说一说你了解的关于lambda函数的全部知识

1) 利用lambda表达式可以编写内嵌的匿名函数，用以替换独立函数或者函数对象；

2) 每当你定义一个lambda表达式后，编译器会自动生成一个匿名类（这个类当然重载了()运算符），我们称为闭包类型（closure type）。那么在运行时，这个lambda表达式就会返回一个匿名的闭包实例，其实一个右值。

所以，我们上面的lambda表达式的结果就是一个个闭包。闭包的一个强大之处是其可以通过传值或者引用的方式捕捉其封装作用域内的变量，前面的方括号就是用来定义捕捉模式以及变量，我们又将其称为lambda捕捉块。

3) lambda表达式的语法定义如下：

```
1 [capture] (parameters) mutable ->return-type {statement};
2 即 [捕获列表](参数)mutable -> 返回值 {函数体}
```

4) lambda必须使用尾置返回来指定返回类型，可以忽略参数列表和返回值，但必须永远包含捕获列表和函数体；

面试宝典

1. 定义 lambda 匿名函数很简单，可以套用如下的语法格式：

```
1 [外部变量访问方式说明符] (参数) mutable noexcept/throw() -> 返回值类型
2 {
3     函数体；
4 }
```

其中各部分的含义分别为：a. **[外部变量访问方式说明符]** `[]` 方括号用于向编译器表明当前是一个 lambda 表达式，其不能被省略。在方括号内部，可以注明当前 lambda 函数的函数体中可以使用哪些“外部变量”。所谓外部变量，指的是和当前 lambda 表达式位于同一作用域内的所有局部变量。b. **(参数)** 和普通函数的定义一样，lambda 匿名函数也可以接收外部传递的多个参数。和普通函数不同的是，如果不需要传递参数，可以连同 `()` 小括号一起省略；c. **mutable** 此关键字可以省略，如果使用则之前的 `()` 小括号将不能省略（参数个数可以为 0）。默认情况下，对于以值传递方式引入的外部变量，不允许在 lambda 表达式内部修改它们的值（可以理解为这部分变量都是 `const` 常量）。而如果想修改它们，就必须使用 **mutable** 关键字。注意：对于以值传递方式引入的外部变量，lambda 表达式修改的是拷贝的那一份，并不会修改真正的外部变量；d. **noexcept/throw()** 可以省略，如果使用，在之前的 `()` 小括号将不能省略（参数个数可以为 0）。默认情况下，lambda 函数的函数体中可以抛出任何类型的异常。而标注 **noexcept** 关键字，则表示函数体内不会抛出任何异常；使用 **throw()** 可以指定 lambda 函数内部可以抛出的异常类型。

e. **-> 返回值类型** 指明 lambda 匿名函数的返回值类型。值得一提的是，如果 lambda 函数体内只有一个 **return** 语句，或者该函数返回 `void`，则编译器可以自行推断出返回值类型，此情况下可以直接省略“-> 返回值类型”。f. **函数体** 和普通函数一样，lambda 匿名函数包含的内部代码都放置在函数体中。该函数体内除了可以使用指定传递进来的参数之外，还可以使用指定的外部变量以及全局范围内的所有全局变量。

```
1 #include <iostream>
2 #include <algorithm>
3 using namespace std;
4 int main()
5 {
6     int num[4] = {4, 2, 3, 1};
7     //对 a 数组中的元素进行排序
8     sort(num, num+4, [](int x, int y) -> bool{ return x < y; });
9     for(int n : num){
10         cout << n << " ";
11     }
12     return 0;
13 }
```

```
14
15  /*
16     程序运行结果:
17         1 2 3 4
18  */
```

我的

简单的示例，展示了如何使用 C++ lambda：

```
1  cpp复制代码#include <iostream>
2
3  int main() {
4      int x = 5;
5      int y = 10;
6
7      // 使用 lambda 表达式定义一个匿名函数对象
8      auto sum = [](int a, int b) -> int {
9          return a + b;
10     };
11
12     // 调用 lambda 表达式
13     int result = sum(x, y);
14
15     std::cout << "Sum: " << result << std::endl;
16
17     return 0;
18 }
```

在上面的示例中，我们定义了一个 lambda 表达式 `sum`，它接受两个整数参数并返回它们的和。然后，我们通过传入变量 `x` 和 `y`，调用这个 lambda 表达式，并将结果存储在 `result` 变量中。最后，我们将结果打印到控制台。

下面是一个示例，展示了一个 lambda 表达式具有捕获列表、参数和返回值的情况：

```
1  #include <iostream>
2
3  int main() {
4      int x = 5;
5      int y = 10;
6
7      // 使用捕获列表、参数和返回值的lambda表达式
8      auto calculate = [x, &y](int a, int b) -> int {
9          y++; // 修改外部变量y的值
10
11          return (x + a) * (y + b);
12     };
13
14     // 调用lambda表达式
```

```

15     int result = calculate(2, 3);
16
17     std::cout << "Result: " << result << std::endl;
18     std::cout << "Updated y: " << y << std::endl;
19
20     return 0;
21 }

```

在上述示例中，我们定义了一个 lambda 表达式 `calculate`，它具有一个捕获列表 `[x, &y]`，以及两个参数 `a` 和 `b`。该 lambda 表达式返回 `(x + a) * (y + b)` 的结果。

在 lambda 函数体内部，我们可以访问捕获的变量 `x`（通过值捕获）和 `y`（通过引用捕获）。我们还修改了外部变量 `y` 的值，这将影响后续的计算。

最后，在 `main` 函数中，我们调用了 lambda 表达式 `calculate`，传入参数 2 和 3，并将结果存储在 `result` 变量中。我们还打印了计算结果和修改后的变量 `y` 的值。

需要注意的是，lambda 表达式的参数列表和返回类型可以根据实际需求进行定义。在示例中，我们指定了两个整数类型的参数，并将返回值设定为整数类型。

总结来说，C++ lambda 表达式可以具有捕获列表、参数和返回值。通过指定捕获方式（值捕获或引用捕获）、参数列表和返回类型，我们可以定义一个灵活的匿名函数，并在其中访问和操作捕获的变量。

11、简述 C++ 右值引用与转移语义

右值引用 (Rvalue Reference) 是 C++11 新标准中引入的新特性，它实现了**转移语义** (Move Semantics) 和**精确传递** (Perfect Forwarding)。它的主要目的有两个方面：

- 消除两个对象交互时不必要的对象拷贝，节省运算存储资源，提高效率
- 能够更简洁明确地定义泛型函数

简单来说，就是右值引用是用来支持转移语义的，转移语义可以将资源（堆、系统对象等）从一个对象转移到另一个对象，**这样能够减少不必要的临时对象的创建、拷贝以及销毁，能够大幅度提高 C++ 应用程序的性能**

在现有的 C++ 机制中，我们可以定义拷贝构造函数和赋值函数。要实现转移语义，需要定义**转移构造函数**，还可以定义**转移赋值操作符**。对于右值的拷贝和赋值会调用转移构造函数和转移赋值操作符。**如果转移构造函数和转移拷贝操作符没有定义，那么就遵循现有的机制，拷贝构造函数和赋值操作符会被调用。**

标准库函数 `move` 可以将一个左值引用**强制转换**成右值引用来使用，相当于 `static_cast`

参考：

- [\[C++\]右值引用和转移语义 stary_yan的博客-CSDN博客](#)
- [C++ 移动构造函数详解](#)

面试宝典

1. 右值引用 一般来说，**不能取地址的表达式，就是右值引用，能取地址的，就是左值。**

```

1 | class A { };
2 | A & r = A(); //error,A()是无名变量, 是右值
3 | A && r = A(); //ok,r是右值引用

```

2. 转移语义 move 本意为 "移动", 但该函数并不能移动任何数据, 它的功能很简单, 就是将**某个左值强制转化为右值**。基于 move() 函数特殊的功能, 其常用于**实现移动语义**。

答案解析

1. **右值引用** C++98/03 标准中就有引用, 使用 "&" 表示。但此种引用方式有一个缺陷, 即正常情况下只能操作 C++ 中的左值, 无法对右值添加引用。举个例子:

```

1 | int num = 10;
2 | int &b = num; //正确
3 | int &c = 10; //错误

```

如上所示, 编译器允许我们为 num 左值建立一个引用, 但不可以为 10 这个右值建立引用。因此, C++98/03 标准中的引用又称为左值引用。

注意: 虽然 C++98/03 标准不支持为右值建立非常量左值引用, 但**允许使用常量左值引用操作右值**。也就是说, **常量左值引用既可以操作左值, 也可以操作右值, 例如:**

```

1 | int num = 10;
2 | const int &b = num;
3 | const int &c = 10;

```

我们知道, **右值往往是没有名称的, 因此要使用它只能借助引用的方式**。这就产生一个问题, 实际开发中我们可能需要对右值进行修改 (实现移动语义时就需要), 显然左值引用的方式是行不通的。

为此, C++11 标准新引入了另一种引用方式, 称为**右值引用**, 用 "&&" 表示。

注意: 和声明左值引用一样, **右值引用也必须立即进行初始化操作, 且只能使用右值进行初始化**, 比如:

```

1 | int num = 10;
2 | //int && a = num; //右值引用不能初始化为左值
3 | int && a = 10; //右值引用初始化为右值

```

和常量左值引用不同的是, **右值引用还可以对右值进行修改**。例如:

```

1 | int && a = 10;
2 | a = 100;
3 | cout << a << endl;
4 | /*    程序运行结果:
5 |      100
6 | */

```

另外值得一提的是, C++ 语法上是支持定义常量右值引用的, 例如:

```

1 | const int&& a = 10; //编译器不会报错

```

但这种定义出来的右值引用并无实际用处。一方面，右值引用主要用于移动语义和完美转发，其中前者需要有修改右值的权限；其次，常量右值引用的作用就是引用一个不可修改的右值，这项工作完全可以交给常量左值引用完成。

2. move语义

```
1 //程序实例
2 #include <iostream>
3 using namespace std;
4 class first {
5 public:
6     first() :num(new int(0)) {
7         cout << "construct!" << endl;
8     }
9     //移动构造函数
10    first(first &&d) :num(d.num) {
11        d.num = NULL;
12        cout << "first move construct!" << endl;
13    }
14 public:    //这里应该是 private, 使用 public 是为了更方便说明问题
15     int *num;
16 };
17 class second {
18 public:
19     second() :fir() {}
20     //用 first 类的移动构造函数初始化 fir
21     second(second && sec) :fir(move(sec.fir)) {
22         cout << "second move construct" << endl;
23     }
24 public:    //这里也应该是 private, 使用 public 是为了更方便说明问题
25     first fir;
26 };
27 int main() {
28     second oth; //输出construct!, 因为构造了一个first
29     second oth2 = move(oth);
30     //cout << *oth.fir.num << endl;    //程序报运行时错误
31     return 0;
32 }
33
34 /*
35     程序运行结果:
36     construct!
37     first move construct!
38     second move construct
39 */
```

14、右值与左值

右值

在C++中，右值（Rvalue）是表达式的一个属性。一个右值表示可以被移动（Move）但不能被复制（Copy）的临时对象或字面量值。

- 1 右值可以分为两类：
- 2 1. 纯右值（Pure Rvalue）：纯右值指的是临时对象、字面量值（如数字、字符串等）和某些运算表达式的结果。
- 3 2. 将亡值（Xvalue, expiring value）：将亡值是指即将被销毁的对象，但它的资源可以被移动到其他地方。通常，具有移动构造函数或移动赋值运算符的对象会创建将亡值。

左值

在C++中，左值（Lvalue）是表达式的一个属性。一个左值表示可以被标识符引用的对象或表达式。

左值的特点如下：

1. 左值具有内存地址，并且可以通过指针来访问。
2. 左值可以出现在赋值操作符的左边或右边。
3. 左值可以进行复制（Copy），因为它们可以被多次使用和修改。

以下是一些示例，展示了左值的使用：

```
1  #include <iostream>
2
3  int main() {
4      int x = 5; // x 是一个左值
5      int y = x; // x 是一个左值, y 是一个左值
6
7      int* ptr = &x; // x 的地址可以获取, 因此 x 是一个左值
8
9      std::cout << "x: " << x << std::endl;
10     std::cout << "y: " << y << std::endl;
11
12     return 0;
13 }
```

在上面的示例中，`x` 和 `y` 都是左值。它们都具有可寻址的内存位置，可以通过指针来间接访问它们。

需要注意的是，虽然从字面上看，`x` 和 `y` 都是整数，但它们在语义上是不同的。`x` 是一个命名的对象，而 `y` 是通过将 `x` 的值复制给 `y` 而创建的新对象。

此外，左值还可以是表达式的结果，只要该表达式返回一个可被标识符引用的对象。例如，`a + b` 是一个左值，如果 `a` 和 `b` 都是左值。

总之，左值表示可以被引用、修改和复制的对象或表达式。它们在C++中起着重要的作用，例如进行赋值操作、传递参数等。

auto用于定义变量，编译器可以自动判断变量的类型。auto主要有以下几种用法：

1. auto的基本使用方法 （1）基本使用语法如下

```
1 | auto name = value; //name 是变量的名字, value 是变量的初始值
```

注意：auto 仅仅是一个占位符，在编译器期间它会被真正的类型所替代。或者说，C++ 中的变量必须是有明确类型的，只是这个类型是由编译器自己推导出来的。

2. auto和 const 的结合使用 （1） auto 与 const 结合的用法 a. 当类型不为引用时，auto 的推导结果将不保留表达式的 const 属性； b. 当类型为引用时，auto 的推导结果将保留表达式的 const 属性。（2）程序实例如下

```
1 | int x = 0;
2 | const auto n = x; //n 为 const int , auto 被推导为 int
3 | auto f = n;      //f 为 const int, auto 被推导为 int (const 属性被抛弃)
4 | const auto &r1 = x; //r1 为 const int& 类型, auto 被推导为 int
5 | auto &r2 = r1;    //r1 为 const int& 类型, auto 被推导为 const int 类型
```

3. 使用auto定义迭代器 在使用 stl 容器的时候，需要使用迭代器来遍历容器里面的元素；**不同容器的迭代器有不同的类型，在定义迭代器时必须指明。**而迭代器的类型有时候比较复杂，请看下面的例子：

16、简述一下 C++11 中的可变参数模板新特性

可变参数模板(variadic template)使得编程者能够创建这样的模板函数和模板类，即可接受可变数量的参数。例如要编写一个函数，它可接受任意数量的参数，参数的类型只需是cout能显示的即可，并将 参数显示为用逗号分隔的列表。

```
1 | int n = 14;
2 | double x = 2.71828;
3 | std::string mr = "Mr.String objects!";
4 | show_list(n, x);
5 | show_list(x*x, '!', 7, mr); //这里的目标是定义show_list()
6 |
7 | /*
8 |     运行结果:
9 |     14, 2.71828
10 |     7.38905, !, 7, Mr.String objects!
11 | */
```

要创建可变参数模板，需要理解几个要点：（1）模板参数包（parameter pack）；（2）函数参数包；（3）展开（unpack）参数包；（4）递归。

44、C++的四种强制转换

reinterpret_cast/const_cast/static_cast /dynamic_cast

reinterpret_cast

```
1 reinterpret_cast<type-id> (expression)
```

type-id 必须是一个指针、引用、算术类型、函数指针或者成员指针。它可以用于类型之间进行强制转换。

const_cast

```
1 const_cast<type-id> (expression)
```

该运算符用来修改类型的const或volatile属性。除了const 或volatile修饰之外， type_id和expression的类型是一样的。用法如下：

- 常量指针被转化成非常量的指针，并且仍然指向原来的对象
- 常量引用被转换成非常量的引用，并且仍然指向原来的对象
- const_cast一般用于修改底指针。如const char *p形式

static_cast < type-id > (expression)

该运算符把expression转换为type-id类型，但没有运行时类型检查来保证转换的安全性。它主要有如下几种用法：

- **用于类层次结构中基类（父类）和派生类（子类）之间指针或引用引用的转换**
 - 进行上行转换（把派生类的指针或引用转换成基类表示）是安全的
 - 进行下行转换（把基类指针或引用转换成派生类表示）时，由于没有动态类型检查，所以是不安全的
- **用于基本数据类型之间的转换**，如把int转换成char，把int转换成enum。这种转换的安全性也要开发人员来保证。
- **把空指针转换成目标类型的空指针**
- **把任何类型的表达式转换成void类型**

注意：static_cast不能转换掉expression的const、volatile、或者__unaligned属性。

dynamic_cast

有类型检查，基类向派生类转换比较安全，但是派生类向基类转换则不太安全

```
1 dynamic_cast <type-id> (expression)
```

该运算符把expression转换成type-id类型的对象。type-id 必须是类的指针、类的引用或者void*

如果 type-id 是类指针类型，那么expression也必须是一个指针，如果 type-id 是一个引用，那么 expression 也必须是一个引用

dynamic_cast运算符可以在执行期决定真正的类型，也就是说expression必须是多态类型。如果下行转换是安全的（也就是说，如果基类指针或者引用确实指向一个派生类对象）这个运算符会传回适当转型过的指针。如果 如果下行转换不安全，这个运算符会传回空指针（也就是说，基类指针或者引用没有指向一个派生类对象）

dynamic_cast主要用于类层次间的上行转换和下行转换，还可以用于类之间的交叉转换

在类层次间进行上行转换时，dynamic_cast和static_cast的效果是一样的

在进行下行转换时，dynamic_cast具有类型检查的功能，比static_cast更安全

举个例子

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  class Base
5  {
6  public:
7      Base() :b(1) {}
8      virtual void fun() {};
9      int b;
10 };
11
12 class Son : public Base
13 {
14 public:
15     Son() :d(2) {}
16     int d;
17 };
18
19 int main()
20 {
21     int n = 97;
22
23     //reinterpret_cast
24     int *p = &n;
25     //以下两者效果相同
26     char *c = reinterpret_cast<char*> (p);
27     char *c2 = (char*)(p);
28     cout << "reinterpret_cast输出: "<< *c2 << endl;
29     //const_cast
30     const int *p2 = &n;
31     int *p3 = const_cast<int*>(p2);
32     *p3 = 100;
33     cout << "const_cast输出: " << *p3 << endl;
34
35     Base* b1 = new Son;
36     Base* b2 = new Base;
37
38     //static_cast
39     Son* s1 = static_cast<Son*>(b1); //同类型转换
40     Son* s2 = static_cast<Son*>(b2); //下行转换, 不安全
41     cout << "static_cast输出: "<< endl;
42     cout << s1->d << endl;
43     cout << s2->d << endl; //下行转换, 原先父对象没有d成员, 输出垃圾值
44
45     //dynamic_cast
46     Son* s3 = dynamic_cast<Son*>(b1); //同类型转换
47     Son* s4 = dynamic_cast<Son*>(b2); //下行转换, 安全
48     cout << "dynamic_cast输出: " << endl;
49     cout << s3->d << endl;
```

```

50     if(s4 == nullptr)
51         cout << "s4指针为nullptr" << endl;
52     else
53         cout << s4->d << endl;
54
55
56     return 0;
57 }
58 //输出结果
59 //reinterpret_cast输出: a
60 //const_cast输出: 100
61 //static_cast输出:
62 //2
63 //-33686019
64 //dynamic_cast输出:
65 //2
66 //s4指针为nullptr

```

从输出结果可以看出，在进行下行转换时，dynamic_cast安全的，如果下行转换不安全的话其会返回空指针，这样在进行操作的时候可以预先判断。而使用static_cast下行转换存在不安全的情况也可以转换成功，但是直接使用转换后的对象进行操作容易造成错误。

46、简述一下 C++11 中四种类型转换

C++中四种类型转换分别为const_cast、static_cast、dynamic_cast、reinterpret_cast，四种转换功能分别如下：

1. **const_cast** 将const变量转为非const
2. **static_cast** 最常用，可以用于各种隐式转换，比如非const转const，static_cast可以用于类向上转换，但向下转换能成功但是不安全。
3. **dynamic_cast** 只能用于含有虚函数的类转换，用于类向上和向下转换 **向上转换：指子类向基类转换。向下转换：指基类向子类转换。** 这两种转换，子类包含父类，当父类转换成子类时可能出现非法内存访问的问题。

dynamic_cast通过判断变量运行时类型和要转换的类型是否相同来判断是否能够进行向下转换。dynamic_cast可以做类之间上下转换，转换的时候会进行类型检查，类型相等成功转换，类型不等转换失败。运用RTTI技术，RTTI是“Runtime Type Information”的缩写，意思是运行时类型信息，它提供了运行时确定对象类型的方法。在c++层面主要体现在dynamic_cast和typeid，vs中虚函数表的-1位置存放了指向type_info的指针，对于存在虚函数的类型，dynamic_cast和typeid都会去查询type_info。

4. **reinterpret_cast** reinterpret_cast可以做任何类型的转换，不过不对转换结果保证，容易出问题。

注意：为什么不用C的强制转换：C的强制转换表面上看起来功能强大什么都能转，但是转换不够明确，不能进行错误检查，容易出错。

2、说说 C++中的智能指针有哪些？ 分别解决的问题以及区别？

1. C++中的智能指针有4种，分别为：shared_ptr、unique_ptr、weak_ptr、auto_ptr，其中 auto_ptr被C++11弃用。
2. 使用智能指针的原因 申请的空间（即new出来的空间），在使用结束时，需要delete掉，否则会形成内存碎片。在程序运行期间，**new出来的对象，在析构函数中delete掉**，但是这种方法不能解决所有问题，因为有时候new发生在某个全局函数里面，该方法会给程序员**造成精神负担**。此时，智能指针就派上了用场。使用智能指针可以很大程度上避免这个问题，**因为智能指针就是一个类，当超出了类的作用域时，类会自动调用析构函数，析构函数会自动释放资源**。所以，智能指针的作用原理就是在函数结束时自动释放内存空间，避免了手动释放内存空间。
3. 四种指针分别解决的问题以及各自特性如下（1）auto_ptr（C++98的方案，C++11已经弃用**采用所有权模式**。

```
1 auto_ptr<string> p1(new string("I reigned loney as a cloud."));
2 auto_ptr<string> p2;
3 p2=p1; //auto_ptr不会报错
```

此时不会报错，p2剥夺了p1的所有权，但是当程序运行时访问p1将会报错。所以auto_ptr的缺点是：存在潜在的内存崩溃问题。

（2）unique_ptr（替换auto_ptr） unique_ptr实现独占式拥有或严格拥有概念，**保证同一时间内只有一个智能指针可以指向该对象**。它对于避免资源泄露，例如，以new创建对象后因为发生异常而忘记调用delete时的情形特别有用。采用所有权模式，和上面例子一样。

```
1 unique_ptr<string> p3(new string("I reigned loney as a cloud."));
2 unique_ptr<string> p4;
3 p4=p3; //此时不会报错
```

编译器认为P4=P3非法，避免了p3不再指向有效数据的问题。因此，unique_ptr比auto_ptr更安全。另外unique_ptr还有更聪明的地方：**当程序试图将一个 unique_ptr 赋值给另一个时，如果源 unique_ptr 是个临时右值，编译器允许这么做；如果源 unique_ptr 将存在一段时间，编译器 将禁止这么做**，比如：

```
1 unique_ptr<string> pu1(new string ("hello world"));
2 unique_ptr<string> pu2;
3 pu2 = pu1; // #1 not allowed
4 unique_ptr<string> pu3;
5 pu3 = unique_ptr<string>(new string ("You")); // #2 allowed
```

其中#1留下悬挂的unique_ptr(pu1)，这可能导致危害。而#2不会留下悬挂的unique_ptr，因为它调用 unique_ptr 的构造函数，该构造函数创建的临时对象在其所有权让给 pu3 后就会被销毁。这种随情况而己的行为表明，unique_ptr 优于允许两种赋值的auto_ptr。

注意：如果确实想执行类似与#1的操作，要安全的重用这种指针，可给它赋新值。**C++有一个 标准库函数 std::move()，让你能够将一个unique_ptr赋给另一个**。例如：

```

1  unique_ptr<string> ps1, ps2;
2  ps1 = demo("hello");
3  ps2 = move(ps1);
4  ps1 = demo("alexia");
5  cout << *ps2 << *ps1 << endl;

```

3) `shared_ptr` (非常好使) `shared_ptr`实现共享式拥有概念。**多个智能指针可以指向相同对象，该对象和其相关资源会在“最后一个引用被销毁”时候释放。**从名字share就可以看出了资源可以被多个指针共享，它使用计数机制来表明资源被几个指针共享。可以通过成员函数**`use_count()`**来查看资源的所有者个数。除了可以通过new来构造，还可以通过传入**`auto_ptr`**, **`unique_ptr`**, **`weak_ptr`**来构造。。

`shared_ptr`是为了解决 `auto_ptr` 在对象所有权上的局限性(`auto_ptr` 是独占的), 在使用引用计数的机制上提供了可以共享所有权的智能指针。

```

1  成员函数:
2  use_count 返回引用计数的个数
3  unique 返回是否是独占所有权( use_count 为 1)
4  swap 交换两个 shared_ptr 对象(即交换所拥有的对象)
5  reset 放弃内部对象的所有权或拥有对象的变更，会引起原有对象的引用计数的减少
6  get 返回内部对象(指针)，由于已经重载了()方法，因此和直接使用对象是一样的.如 shared_ptr
7  sp(new int(1)); sp 与 sp.get()是等价的

```

4) `weak_ptr` **`weak_ptr` 是一种不控制对象生命周期的智能指针，它指向一个 `shared_ptr` 管理的对象。进行该对象的内存管理的是那个强引用的 `shared_ptr`。**`weak_ptr`只是提供了对管理对象的一个访问手段。`weak_ptr`设计的目的是为配合 `shared_ptr` 而引入的一种智能指针来协助 `shared_ptr` 工作，它只可以从一个 `shared_ptr` 或另一个 **`weak_ptr` 对象构造**，它的构造和析构不会引起引用计数的增加或减少。`weak_ptr`是用来解决`shared_ptr`**相互引用时的死锁问题**，如果说两个`shared_ptr`相互引用，那么这两个指针的引用计数永远不可能下降为0,资源永远不会释放。它是**对对象的一种弱引用，不会增加对象的引用计数**，和`shared_ptr`之间可以相互转化，`shared_ptr`可以直接赋值给它，它可以通过调用**`lock`**函数来获得`shared_ptr`。

```

1  class B;
2  class A
3  {
4  public:
5      shared_ptr<B> pb_;
6      ~A()
7      {
8          cout<<"A delete\n";
9      }
10 };
11 class B
12 {
13 public:
14     shared_ptr<A> pa_;
15     ~B()
16     {
17         cout<<"B delete\n";
18     }
19 };
20 void fun()
21 {

```

```

22     shared_ptr<B> pb(new B());
23     shared_ptr<A> pa(new A());
24     pb->pa_ = pa;
25     pa->pb_ = pb;
26     cout<<pb.use_count()<<endl;
27     cout<<pa.use_count()<<endl;
28 }
29 int main()
30 {
31     fun();
32     return 0;
33 }

```

可以看到fun函数中pa，pb之间互相引用，两个资源的引用计数为2，当要跳出函数时，智能指针pa，pb析构时两个资源引用计数会减一，但是两者引用计数还是为1，导致跳出函数时资源没有被释放（A B的析构函数没有被调用），如果把其中一个改为weak_ptr就可以了，我们把类A里面的shared_ptr pb_；改为weak_ptr pb；运行结果如下，这样的话，资源B的引用开始就只有1，当pb析构时，B的计数变为0，B得到释放，B释放的同时也会使A的计数减一，同时pa析构时使A的计数减一，那么A的计数为0，A得到释放。

```

1  注意：我们不能通过weak_ptr直接访问对象的方法，比如B对象中有一个方法print()，我们不能
2  这样访问，pa->pb->print()；英文pb是一个weak_ptr，应该先把它转化为shared_ptr，如：
3  shared_ptr p = pa->pb_.lock()；p->print()；

```

2、简述 C++ 中智能指针的特点

1. C++中的智能指针有4种，分别为：**shared_ptr**、**unique_ptr**、**weak_ptr**、**auto_ptr**，其中 auto_ptr被C++11弃用。
2. **为什么要使用智能指针**：智能指针的作用是管理一个指针，因为存在申请的空间在函数结束时忘记释放，造成**内存泄漏**的情况。使用智能指针可以很大程度上避免这个问题，因为智能指针就是一个类，当**超出了类的作用域时**，类会**自动调用析构函数，自动释放资源**。
3. 四种指针各自特性（1）**auto_ptr 所有权模式**，auto指针存在的问题是，两个智能指针同时指向一块内存，就会两次释放同一块资源，自然报错。（2）**unique_ptr** unique指针规定一个智能指针独占一块内存资源。当两个智能指针同时指向一块内存，编译报错。实现原理：**将拷贝构造函数和赋值拷贝构造函数申明为private或delete**。不允许拷贝构造函数和赋值操作符，但是**支持移动构造函数，通过std::move把一个对象指针变成右值之后可以移动给另一个unique_ptr**（3）**shared_ptr 共享指针**可以实现多个智能指针指向相同对象，该对象和其相关资源会在引用为0时被销毁释放。实现原理：有一个引用计数的指针类型变量，专门用于引用计数，**使用拷贝构造函数和赋值拷贝构造函数时，引用计数加1，当引用计数为0时，释放资源**。注意：weak_ptr、shared_ptr存在一个问题，当两个shared_ptr指针相互引用时，那么这两个指针的引用计数不会下降为0，资源得不到释放。因此引入weak_ptr，weak_ptr是弱引用，weak_ptr的构造和析构不会引起引用计数的增加或减少。

3、智能指针的原理、常用的智能指针及实现

原理

智能指针是一个类，用来存储指向动态分配对象的指针，负责自动释放动态分配的对象，防止堆内存泄漏。动态分配的资源，交给一个类对象去管理，当类对象声明周期结束时，自动调用析构函数释放资源

常用的智能指针

(1) shared_ptr

shared_ptr是引用计数型智能指针，一个类模板（class template），它只有一个类型参数

实现原理：采用引用计数器的方法，允许多个智能指针指向同一个对象，每当多一个指针指向该对象时，指向该对象的所有智能指针内部的引用计数加1，每当减少一个智能指针指向对象时，引用计数会减1，当计数为0的时候会自动的释放动态分配的资源。

- 智能指针将一个计数器与类指向的对象相关联，引用计数器跟踪共有多少个类对象共享同一指针
- 每次创建类的新对象时，初始化指针并将引用计数置为1
- 当对象作为另一对象的副本而创建时，拷贝构造函数拷贝指针并增加与之相应的引用计数
- 对一个对象进行赋值时，赋值操作符减少左操作数所指对象的引用计数（如果引用计数为减至0，则删除对象），并增加右操作数所指对象的引用计数
- 调用析构函数时，构造函数减少引用计数（如果引用计数减至0，则删除基础对象）

(2) unique_ptr

unique_ptr采用的是独享所有权语义，一个非空的unique_ptr总是拥有它所指向的资源。转移一个unique_ptr将会把所有权全部从源指针转移给目标指针，源指针被置空；所以unique_ptr不支持普通的拷贝和赋值操作，不能用在STL标准容器中；局部变量的返回值除外（因为编译器知道要返回的对象将要被销毁）；如果你拷贝一个unique_ptr，那么拷贝结束后，这两个unique_ptr都会指向相同的资源，造成在结束时对同一内存指针多次释放而导致程序崩溃。

(3) weak_ptr

weak_ptr也是一个引用计数型智能指针，但是它不增加对象的引用次数，即弱（weak）引用

weak_ptr：弱引用。引用计数有一个问题就是互相引用形成环（环形引用），这样两个指针指向的内存都无法释放。需要使用weak_ptr打破环形引用。weak_ptr是一个弱引用，它是为了配合shared_ptr而引入的一种智能指针，它指向一个由shared_ptr管理的对象而不影响所指对象的生命周期，也就是说，它只引用，不计数。如果一块内存被shared_ptr和weak_ptr同时引用，当所有shared_ptr析构了之后，不管还有没有weak_ptr引用该内存，内存也会被释放。所以weak_ptr不保证它指向的内存一定是有效的，在使用之前使用函数lock()检查weak_ptr是否为空指针。

(4) auto_ptr

主要是为了解决“有异常抛出时发生内存泄漏”的问题。因为发生异常而无法释放内存。

auto_ptr有拷贝语义，拷贝后源对象变得无效，这可能引发很严重的问题；而unique_ptr则无拷贝语义，但提供了移动语义，这样的错误不再可能发生，因为很明显必须使用std::move()进行转移。

auto_ptr不支持拷贝和赋值操作，不能用在STL标准容器中。STL容器中的元素经常要支持拷贝、赋值操作，在这过程中auto_ptr会传递所有权，所以不能在STL中使用。

智能指针shared_ptr代码实现：

```
1  template<typename T>
2  class SharedPtr
3  {
4  public:
5      SharedPtr(T* ptr = NULL):_ptr(ptr), _pcount(new int(1))
6      {}
```

```

7
8     SharedPtr(const SharedPtr& s):_ptr(s._ptr), _pcount(s._pcount){
9         (*_pcount)++;
10    }
11
12    SharedPtr<T>& operator=(const SharedPtr& s){
13        if (this != &s)
14        {
15            if (--(*this->_pcount) == 0)
16            {
17                delete this->_ptr;
18                delete this->_pcount;
19            }
20            _ptr = s._ptr;
21            _pcount = s._pcount;
22            (*_pcount)++;
23        }
24        return *this;
25    }
26    T& operator*()
27    {
28        return *(this->_ptr);
29    }
30    T* operator->()
31    {
32        return this->_ptr;
33    }
34    ~SharedPtr()
35    {
36        --(*this->_pcount);
37        if (*this->_pcount == 0)
38        {
39            delete _ptr;
40            _ptr = NULL;
41            delete _pcount;
42            _pcount = NULL;
43        }
44    }
45 private:
46     T* _ptr;
47     int* _pcount; //指向引用计数的指针
48 };

```

总结

- shared_ptr控制对象的生命期。shared_ptr是强引用（想象成用铁丝 绑住堆上的对象），只要有一个指向x对象的shared_ptr存在，该x对象 就不会析构。当指向对象x的**最后一个shared_ptr析构或reset()的时候，x 保证会被销毁**
- weak_ptr**不控制对象的生命期，但是它知道对象是否还活着**。如果对象还活着，那么它可以提升（promote）为有效的shared_ptr；如果对象已经死了，提升会失败，返回一个空的shared_ptr。“提升 / lock()”行为是线程安全的

- shared_ptr/weak_ptr的“计数”在主流平台上是原子操作，没有用锁，性能不俗
- shared_ptr/weak_ptr的线程安全级别与std::string和STL容器一样，后面还会讲

4、智能指针的原理

智能指针是C++中用于管理动态分配资源的一种机制，它是一个类，用来存储指向动态分配对象的指针（封装一个原始指针），负责自动释放动态分配的对象，防止堆内存泄漏。动态分配的资源，交给一个类对象去管理，当类对象声明周期结束时，自动调用析构函数释放资源。

智能指针的主要原理是利用了RAII（Resource Acquisition Is Initialization）的概念，即资源获取即初始化。

5、手写一个shared_ptr

首先回顾一下shared_ptr的概念

- shared_ptr 内部维护了一个引用计数器，记录有多少个 shared_ptr 共享同一块内存资源。每当创建一个新的 shared_ptr 对象指向该资源时，引用计数就加一；当 shared_ptr 对象被销毁或者重新赋值时，引用计数就减一。只有当引用计数变为零时，表示没有任何 shared_ptr 指向该资源，内存才会被释放。
- shared_ptr将一个计数器与类指向的对象相关联，引用计数跟踪该类有多少个对象共享同一指针。每次创建类的新对象时，初始化指针并将引用计数置为1；
- 当对象作为另一对象的副本而创建时，拷贝构造函数拷贝指针并增加与之相应的引用计数；
- 对一个对象进行赋值时，赋值操作符减少左操作数所指对象的引用计数（如果引用计数为减至0，则删除对象），并增加右操作数所指对象的引用计数；
- 调用析构函数时，构造函数减少引用计数（如果引用计数减至0，则自动删除所指向的堆内存）。

智能指针shared_ptr代码实现

```
1  template<typename T>
2  class SharedPtr
3  {
4  public:
5      // 默认构造函数，创建一个空的 SharedPtr 对象。
6      explicit SharedPtr();
7      // 接受一个原始指针 _ptr，创建一个包装该指针的 SharedPtr 对象
8      explicit SharedPtr(T *_ptr);
9      // 拷贝构造函数，用于初始化一个新的 SharedPtr 对象，与另一个 SharedPtr 共享所管理的对象。
10     SharedPtr(const SharedPtr& p);
11     // 析构函数，负责释放所管理的对象，并递减计数器。当计数器为零时，表示没有任何 SharedPtr 对象引
        用该对象，可以安全删除。
12     ~SharedPtr();
13
14     // 重载拷贝赋值运算符，用于将另一个 SharedPtr 的状态赋值给当前对象。
15     // 在赋值之前，会先递减当前对象的计数器，并根据需要释放资源。
16     // 然后将另一个 SharedPtr 的指针和计数器赋值给当前对象。
```

```

17     SharedPtr& operator=(const SharedPtr& p);
18
19     // 重载解引用运算符，返回所管理对象的引用。
20     T& operator*();
21     // 重载箭头运算符，返回所管理对象的指针。
22     T* operator->();
23     // 重载布尔值操作
24     operator bool();
25
26     // 返回所管理的原始指针。
27     T* get() const;
28
29     // 返回当前共享所管理对象的 SharedPtr 数量。
30     size_t use_count();
31
32     // 检查是否只有一个 SharedPtr 引用该对象
33     bool unique();
34
35     // 交换两个 SharedPtr 对象之间的资源和状态
36     void swap(SharedPtr& p);
37
38 private:
39     size_t *count;
40     T *ptr;
41 };
42
43 template<typename T>
44 SharedPtr<T>::SharedPtr() : count(new size_t(0)), ptr(nullptr)
45 {}
46
47 template<typename T>
48 SharedPtr<T>::SharedPtr(T *_ptr) : count(new size_t(1)), ptr(_ptr)
49 {}
50
51 template<typename T>
52 SharedPtr<T>::~SharedPtr()
53 {
54     --(*count);
55     if(*count <= 0) {
56         delete ptr;
57         delete count;
58         ptr = nullptr;
59         count = nullptr;
60     }
61     std::cout << "shared ptr destory." << std::endl;
62 }
63
64 template<typename T>
65 SharedPtr<T>::SharedPtr(const SharedPtr &p)
66 {
67     count = p.count;
68     ptr = p.ptr;
69     ++(*count);

```

```

70 }
71
72 template<typename T>
73 SharedPtr<T>& SharedPtr<T>::operator=(const SharedPtr &p)
74 {
75     // 如果是原指针, 直接返回
76     if(ptr == p.ptr) {
77         return *this;
78     }
79
80     if(ptr) {
81         --(*count);
82         if((*count) == 0) {
83             delete ptr;
84             delete count;
85         }
86     }
87
88     ptr = p.ptr;
89     count = p.count;
90     ++(*count);
91     return *this;
92 }
93
94 template<typename T>
95 T& SharedPtr<T>::operator*()
96 {
97     return *ptr;
98 }
99
100 template<typename T>
101 T* SharedPtr<T>::operator->()
102 {
103     return ptr;
104 }
105
106 template<typename T>
107 SharedPtr<T>::operator bool()
108 {
109     return ptr != nullptr;
110 }
111
112 template<typename T>
113 T* SharedPtr<T>::get() const
114 {
115     return ptr;
116 }
117
118 template<typename T>
119 size_t SharedPtr<T>::use_count()
120 {
121     return *count;
122 }

```

```

123
124 template<typename T>
125 bool SharedPtr<T>::unique()
126 {
127     return *count == 1;
128 }
129
130 template<typename T>
131 void SharedPtr<T>::swap(SharedPtr& p)
132 {
133     std::swap(*this, p);
134 }

```

为什么shared_ptr赋值时，左边引用计数减一，右边引用计数加一

在 `shared_ptr` 赋值时，左边引用计数减一，右边引用计数加一的原因是为了维护资源的正确引用计数。

当一个 `shared_ptr` 对象赋值给另一个 `shared_ptr` 对象时，目标对象（左边）可能已经指向了其他资源。在进行赋值操作之前，我们希望递减左边对象的引用计数，以避免资源的过早释放。

然后，我们将右边对象的资源和引用计数复制到左边对象中。递增右边对象的引用计数是因为赋值操作后，左右两个对象都指向了同一块资源，需要确保引用计数的正确性。

这样做的好处是，在赋值完成后，左右两个 `shared_ptr` 对象都可以独立地管理资源，并且其引用计数也能正确地反映出资源被引用的次数。同时，如果左边对象的引用计数减少到零，则表示没有任何对象引用该资源，可以安全地释放资源。

这种引用计数的机制使得多个 `shared_ptr` 对象能够共享同一个资源，并自动在最后一个引用对象销毁时，释放内存。但是需要注意避免循环引用的情况，即两个或多个 `shared_ptr` 对象相互引用导致无法释放资源的问题。

6、手写实现智能指针类需要实现哪些函数？

1、智能指针是一个数据类型，一般用模板实现，模拟指针行为的同时还提供自动垃圾回收机制。它会自动记录 `SmartPointer<T*>` 对象的引用计数，一旦 `T` 类型对象的引用计数为0，就释放该对象。

除了指针对象外，我们还需要一个引用计数的指针设定对象的值，并将引用计数计为1，需要一个构造函数。新增对象还需要一个构造函数，析构函数负责引用计数减少和释放内存。

通过覆写赋值运算符，才能将一个旧的智能指针赋值给另一个指针，同时旧的引用计数减1，新的引用计数加1

2、一个构造函数、拷贝构造函数、赋值构造函数、析构函数、移动函数；

7、智能指针shared_ptr出现循环引用怎么解决

弱指针用于专门解决shared_ptr循环引用的问题，weak_ptr不会修改引用计数，即其存在与否并不影响对象的引用计数器。循环引用就是：两个对象互相使用一个shared_ptr成员变量指向对方。弱引用并不对对象的内存进行管理，在功能上类似于普通指针，然而一个比较大的区别是，弱引用能检测到所管理的对象是否已经被释放，从而避免访问非法内存。

我的：

在C++中，当使用 `std::shared_ptr` 来管理资源时，循环引用（cyclic reference）可能会导致资源无法释放，从而造成内存泄漏。循环引用指的是两个或多个对象彼此持有对方的 `std::shared_ptr`，从而导致资源无法被正确释放。

为了解决 `std::shared_ptr` 循环引用问题，可以采用以下几种方法：

1. 使用std::weak_ptr： `std::weak_ptr` 是 `std::shared_ptr` 的一种弱引用，它不会增加所指对象的引用计数，也不拥有所指对象。通过将其中一个 `std::shared_ptr` 转换为 `std::weak_ptr` 来打破循环引用。

```
1  cpp复制代码class ClassB; // 前向声明
2
3  class ClassA {
4      std::shared_ptr<ClassB> ptrB;
5  };
6
7  class ClassB {
8      std::weak_ptr<ClassA> weakPtrA;
9  };
```

使用 `std::weak_ptr` 可以避免循环引用导致资源无法释放，但在使用 `std::weak_ptr` 时需要注意判断其是否过期（expired），可以通过调用 `std::weak_ptr` 的 `lock()` 函数获取一个有效的 `std::shared_ptr`，然后再进行操作。

面试宝典

为了解决循环引用导致的内存泄漏，引入了弱指针weak_ptr，weak_ptr的构造函数**不会修改引用计数的值，从而不会对对象的内存进行管理**，其类似一个普通指针，但是不会指向引用计数的共享内存，**但是可以检测到所管理的对象是否已经被释放，从而避免非法访问。**

例子

shared_ptr是带引用计数的智能指针，可以说大部分的情形选择用shared_ptr不会出问题。那么weak_ptr是什么，应该怎么用呢？

weak_ptr也是智能指针，但是比较弱，感觉没什么用。其实它的出现是伴随shared_ptr而来，尤其是解决了一个引用计数导致的问题：在存在循环引用的时候会出现内存泄漏。

关于循环引用，看下面这个小例子就足够了：

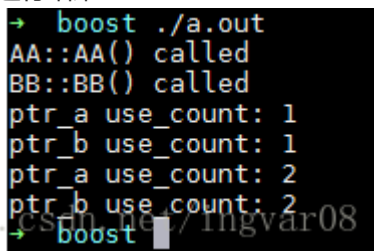
```
1  #include <iostream>
2  #include <boost/smart_ptr.hpp>
3  using namespace std;
```

```

4  using namespace boost;
5
6  class BB;
7  class AA
8  {
9  public:
10     AA() { cout << "AA::AA() called" << endl; }
11     ~AA() { cout << "AA::~~AA() called" << endl; }
12     shared_ptr<BB> m_bb_ptr; //!
13 };
14
15 class BB
16 {
17 public:
18     BB() { cout << "BB::BB() called" << endl; }
19     ~BB() { cout << "BB::~~BB() called" << endl; }
20     shared_ptr<AA> m_aa_ptr; //!
21 };
22
23 int main()
24 {
25     shared_ptr<AA> ptr_a (new AA);
26     shared_ptr<BB> ptr_b ( new BB);
27     cout << "ptr_a use_count: " << ptr_a.use_count() << endl;
28     cout << "ptr_b use_count: " << ptr_b.use_count() << endl;
29     //下面两句导致了AA与BB的循环引用，结果就是AA和BB对象都不会析构
30     ptr_a->m_bb_ptr = ptr_b;
31     ptr_b->m_aa_ptr = ptr_a;
32     cout << "ptr_a use_count: " << ptr_a.use_count() << endl;
33     cout << "ptr_b use_count: " << ptr_b.use_count() << endl;
34 }

```

运行结果:



```

→ boost ./a.out
AA::AA() called
BB::BB() called
ptr_a use_count: 1
ptr_b use_count: 1
ptr_a use_count: 2
ptr_b use_count: 2
→ boost

```

可以看到由于AA和BB内部的shared_ptr各自保存了对方的一次引用，所以导致了ptr_a和ptr_b销毁的时候都认为内部保存的指针计数没有变成0，所以AA和BB的析构函数不会被调用。解决方法就是把一个shared_ptr替换成weak_ptr。

```

1  #include <iostream>
2  #include <boost/smart_ptr.hpp>
3  using namespace std;
4  using namespace boost;
5
6  class BB;
7  class AA
8  {
9  public:

```

```

10     AA() { cout << "AA::AA() called" << endl; }
11     ~AA() { cout << "AA::~~AA() called" << endl; }
12     weak_ptr<BB> m_bb_ptr; //!
13 };
14
15 class BB
16 {
17 public:
18     BB() { cout << "BB::BB() called" << endl; }
19     ~BB() { cout << "BB::~~BB() called" << endl; }
20     shared_ptr<AA> m_aa_ptr; //!
21 };
22
23 int main()
24 {
25     shared_ptr<AA> ptr_a (new AA);
26     shared_ptr<BB> ptr_b ( new BB);
27     cout << "ptr_a use_count: " << ptr_a.use_count() << endl;
28     cout << "ptr_b use_count: " << ptr_b.use_count() << endl;
29     //下面两句导致了AA与BB的循环引用，结果就是AA和BB对象都不会析构
30     ptr_a->m_bb_ptr = ptr_b;
31     ptr_b->m_aa_ptr = ptr_a;
32     cout << "ptr_a use_count: " << ptr_a.use_count() << endl;
33     cout << "ptr_b use_count: " << ptr_b.use_count() << endl;
34 }

```

运行结果：

```

→ boost ./a.out
AA::AA() called
BB::BB() called
ptr_a use_count: 1
ptr_b use_count: 1
ptr_a use_count: 2
ptr_b use_count: 1
BB::~~BB() called
AA::~~AA() called
→ boost

```

关于[weak_ptr](#)更详细的说明可以阅读[boost](#)的文档，绝对的宝库。

8、强弱智能指针

- **强智能指针**：资源每被强智能指针引用一次，引用计数+1，释放引用计数-1，如shared_ptr;
- **弱智能指针**：仅仅起到观察作用，观察对象释放还存在，不改变资源的引用计数，如weak_ptr.

12、boost::scope_ptr智能指针（使用及原理分析）

boost::scoped_ptr是一个比较简单的智能指针，它能保证在**离开作用域**之后它所管理对象能被**自动释放**。下面这个例子将介绍它的使用：

```
1 1 #include <iostream>
2 2 #include <boost/scoped_ptr.hpp>
3 3
4 4 using namespace std;
5 5 class Book {
6 6 public:
7 7     Book()
8 8 {
9 9         cout << "Creating book ..." << endl;
10 10 }
11 11
12 12     ~Book()
13 13 {
14 14         cout << "Destroying book ..." << endl;
15 15 }
16 16 };
17 17
18 18 int main()
19 19 {
20 20     cout << "====Main Begin====" << endl;
21 21 {
22 22         boost::scoped_ptr<Book> myBook(new Book());
23 23 }
24 24     cout << "==== Main End =====" << endl;
25 25
26 26     return 0;
27 27 }
```

运行结果：

```
====Main Begin====
Creating book ...
Destroying book ...
==== Main End =====
请按任意键继续. . .
```

可以看出：当myBook离开了它的作用域之后，它所管理的Book对象也随之销毁。

特点——不能共享控制权

scoped_ptr不能通过其他scoped_ptr共享控制权，因为在scoped_ptr类的内部将**拷贝构造函数**和**=运算符重载**定义为**私有的**。我们看下scoped_ptr类的定义就清楚了：

```
1 namespace boost
2 {
3     // 声明一个模板类 scoped_ptr, 继承 noncopyable, 用于管理指针资源
4     template<typename T> class scoped_ptr : noncopyable
5     {
6     private:
```



```

7      T *px; // 指向被管理的对象的指针
8
9      // 复制构造函数和赋值操作符被私有化, 禁止拷贝和赋值
10     scoped_ptr(scoped_ptr const &);
11     scoped_ptr &operator=(scoped_ptr const &);
12
13     typedef scoped_ptr<T> this_type;
14
15     void operator==( scoped_ptr const & ) const; // 禁用比较操作符==
16     void operator!=( scoped_ptr const & ) const; // 禁用比较操作符!=
17 public:
18     // 构造函数: 接受指针参数, 并创建 scoped_ptr 实例
19     explicit scoped_ptr(T *p = 0);
20     // 析构函数: 释放指针指向的对象资源
21     ~scoped_ptr();
22
23     // 构造函数: 接受 std::auto_ptr 参数, 并创建 scoped_ptr 实例
24     explicit scoped_ptr( std::auto_ptr<T> p ): px( p.release() );
25     // 重置指针指向的对象
26     void reset(T *p = 0);
27
28     // 解引用操作符: 返回指针指向的对象的引用
29     T &operator*() const;
30     // 成员访问操作符: 返回指针指向的对象的指针
31     T *operator->() const;
32     // 获取内部的指针
33     T *get() const;
34
35     // 交换两个 scoped_ptr 实例的指针
36     void swap(scoped_ptr &b);
37 };
38
39 // 交换两个 scoped_ptr 实例的指针
40 template<typename T>
41 void swap(scoped_ptr<T> &a, scoped_ptr<T> &b);
42 }
43

```

下面这段代码中的注释部分打开会造成编译失败:

```

1  #include <iostream>
2  #include <boost/scoped_ptr.hpp>
3
4  using namespace std;
5
6  class Book
7  {
8  public:
9      Book()
10     {
11         cout << "Creating book ..." << endl;
12     }
13

```

```

14     ~Book()
15     {
16         cout << "Destroying book ..." << endl;
17     }
18 };
19
20 int main()
21 {
22     cout << "====Main Begin====" << endl;
23     {
24         boost::scoped_ptr<Book> myBook(new Book());
25         //boost::scoped_ptr<Book> myBook1(myBook);    // Error: scoped_ptr的拷贝构造函数
私有
26         //boost::scoped_ptr<Book> myBook2 = myBook;    // Error: scoped_ptr的=运算符重载
私有
27     }
28     cout << "==== Main End =====" << endl;
29
30     return 0;
31 }

```

所以，scoped_ptr不能用在标准库的容器中，因为容器中的push_back操作需要调用scoped_ptr的=运算符重载函数，结果就是会导致编译失败。

```

1  #include <iostream>
2  #include <string>
3  #include <vector>
4  #include <boost/scoped_ptr.hpp>
5
6  using namespace std;
7
8  class Book
9  {
10 private:
11     string name_;
12
13 public:
14     Book(string name) : name_(name)
15     {
16         cout << "Creating book " << name_ << " ..." << endl;
17     }
18
19     ~Book()
20     {
21         cout << "Destroying book " << name_ << " ..." << endl;
22     }
23 };
24
25 int main()
26 {
27     cout << "====Main Begin====" << endl;
28     {
29         boost::scoped_ptr<Book> myBook(new Book(" [1984] "));

```

```

30     vector<boost::scoped_ptr<Book>> vecScoped;
31     //vecScoped.push_back(myBook);    // Error: push_back操作内部调用了scoped_ptr的=运
算符重载函数
32 }
33     cout << "==== Main End =====> << endl;
34
35     return 0;
36 }

```

编译检查=万无一失?

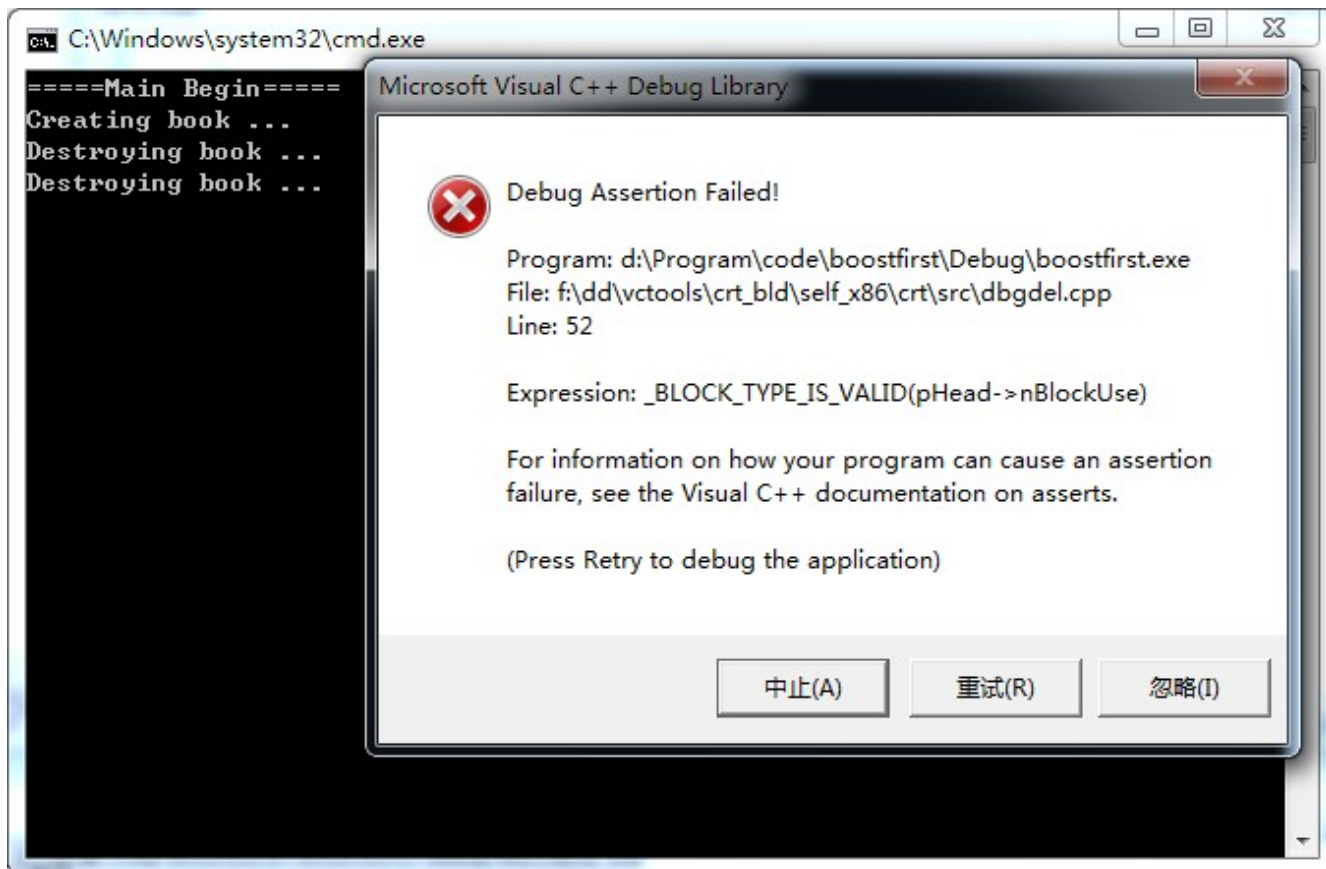
虽然我们无法通过scoped_ptr的拷贝构造函数和=运算符重载函数共享控制权。那如果将一个对象交给多个scoped_ptr来管理会怎样?

```

1  #include <iostream>
2  #include <boost/scoped_ptr.hpp>
3
4  using namespace std;
5
6  class Book
7  {
8  public:
9      Book()
10     {
11         cout << "Creating book ..." << endl;
12     }
13
14     ~Book()
15     {
16         cout << "Destroying book ..." << endl;
17     }
18 };
19
20 int main()
21 {
22     cout << "====Main Begin====> << endl;
23     {
24         Book * book = new Book();
25         boost::scoped_ptr<Book> myBook(book);
26         boost::scoped_ptr<Book> myBook1(book);
27     }
28     cout << "==== Main End =====> << endl;
29
30     return 0;
31 }

```

我们发现编译没报错，但是运行时出错了，如下：



之所以会这样是因为每个scoped_ptr对象都保存了自己所管理对象指针px，scoped_ptr对象在离开自己作用域时会调用了自身的析构函数，在析构函数内部会调用delete px，当多个scoped_ptr管理同一个对象时，那么在它们离开作用域之后，势必会多次调用delete以释放它们所管理的对象，从而造成程序运行出错。

其他接口

- 1 虽然scoped_ptr不能转移控制权，但是它们可以交换共享权。就以下的代码举个例子：

```
1  #include <iostream>
2  #include <string>
3  #include <boost/scoped_ptr.hpp>
4
5  using namespace std;
6
7  class Book
8  {
9  private:
10     string name_;
11
12 public:
13     Book(string name) : name_(name)
14     {
15         cout << "Creating book " << name_ << " ..." << endl;
16     }
17
18     ~Book()
19     {
20         cout << "Destroying book " << name_ << " ..." << endl;
```

```

21     }
22 };
23
24 int main()
25 {
26     cout << "====Main Begin====" << endl;
27     {
28         boost::scoped_ptr<Book> myBook(new Book("「1984」"));
29         boost::scoped_ptr<Book> myBook1(new Book("「A Song of Ice and Fire」"));
30         myBook.swap(myBook1);
31     }
32     cout << "==== Main End =====" << endl;
33
34     return 0;
35 }

```

运行结果：

```

====Main Begin====
Creating book 「1984」 ...
Creating book 「A Song of Ice and Fire」 ...
Destroying book 「1984」 ...
Destroying book 「A Song of Ice and Fire」 ...
==== Main End =====
请按任意键继续. . .

```

根据栈的特性，应该是后面构造的scoped_ptr对象先销毁（从而销毁了它们所管理的对象），正是因为我们两个智能指针的控制权进行交换之后，才出现了这种相反的结果。

此外，在scoped_ptr离开作用域之前也是可以显式销毁它们所管理的对象的。调用它的reset方法即可。请看下面例子：

```

1  #include <iostream>
2  #include <string>
3  #include <boost/scoped_ptr.hpp>
4
5  using namespace std;
6
7  class Book
8  {
9  private:
10     string name_;
11
12 public:
13     Book(string name) : name_(name)
14     {
15         cout << "Creating book " << name_ << " ..." << endl;
16     }
17
18     ~Book()
19     {
20         cout << "Destroying book " << name_ << " ..." << endl;
21     }
22 };

```

```

23
24 int main()
25 {
26     cout << "====Main Begin====" << endl;
27     {
28         boost::scoped_ptr<Book> myBook(new Book("「1984」"));
29         myBook.reset();
30         cout << "After reset ..." << endl;
31     }
32     cout << "==== Main End =====" << endl;
33
34     return 0;
35 }

```

运行结果：

```

====Main Begin====
Creating book 「1984」 ...
Destroying book 「1984」 ...
After reset ...
==== Main End =====
请按任意键继续. . .

```

可以看出：程序在输出“After reset ...”之前已经完成了对所管理对象的释放。

总结（摘自《超越C++标准库：Boost库导论》）

使用裸指针来写异常安全和无错误的代码是很复杂的。使用智能指针来自动地把动态分配对象的生存期限限制在一个明确的范围之内，是解决这种问题的一个有效的方法，并且提高了代码的可读性、可维护性和质量。scoped_ptr明确地表示被指物不能被共享和转移。当一个动态分配的对象被传送给 scoped_ptr，它就成为了这个对象的唯一的拥有者。因为scoped_ptr几乎总是以自动变量或数据成员来分配的，因此它可以在离开作用域时正确地销毁，从而在执行流由于返回语句或异常抛出而离开作用域时，总能释放它所管理的内存。

在以下情况时使用scoped_ptr：

- 在可能有异常抛出的作用域里使用指针
- 函数里有几条控制路径
- 动态分配对象的生存期应被限制于特定的作用域内
- 异常安全非常重要时(始终如此!)

参考

- <http://www.cnblogs.com/sld666666/archive/2010/12/16/1908265.html>
- Björn Karlsson: Beyond the C++ Standard Library: An Introduction to Boost（《超越C++标准库：Boost库导论》）

（完）

本文转自 <https://www.cnblogs.com/helloamigo/p/3572533.html>，如有侵权，请联系删除。

14、C++智能指针详解

<http://t.csdn.cn/36CE8>

参考资料：《C++ Primer中文版 第五版》

我们知道除了静态内存和栈内存外，每个程序还有一个内存池，这部分内存被称为自由空间或者堆。程序用堆来存储动态分配的对象即那些在程序运行时分配的对象，当动态对象不再使用时，我们的代码必须显式的销毁它们。

在C++中，动态内存的管理是用一对运算符完成的：new和delete，new:在动态内存中为对象分配一块空间并返回一个指向该对象的指针，delete：指向一个动态独享的指针，销毁对象，并释放与之关联的内存。

动态内存管理经常会出现两种问题：一种是忘记释放内存，会造成内存泄漏；一种是尚有指针引用内存的情况下就释放了它，就会产生引用非法内存的指针。

为了更加容易（更加安全）的使用动态内存，引入了智能指针的概念。智能指针的行为类似常规指针，重要的区别是它负责自动释放所指向的对象。标准库提供的两种智能指针的区别在于管理底层指针的方法不同，**shared_ptr允许多个指针指向同一个对象**，**unique_ptr则“独占”所指向的对象**。标准库还定义了一种名为weak_ptr的伴随类，它是一种弱引用，指向shared_ptr所管理的对象，这三种智能指针都定义在memory头文件中。

shared_ptr类

创建智能指针时必须提供额外的信息，指针可以指向的类型：

```
1 shared_ptr<string> p1;  
2 shared_ptr<list<int>> p2;
```

默认初始化的智能指针中保存着一个空指针。

智能指针的使用方式和普通指针类似，解引用一个智能指针返回它指向的对象，在一个条件判断中使用智能指针就是检测它是不是空。

```
1 if(p1 && p1->empty())  
2     *p1 = "hi";
```

如下表所示是shared_ptr和unique_ptr都支持的操作：

表 12.1: shared_ptr 和 unique_ptr 都支持的操作	
shared_ptr<T> sp	空智能指针，可以指向类型为 T 的对象
unique_ptr<T> up	
p	将 p 用作一个条件判断，若 p 指向一个对象，则为 true
*p	解引用 p，获得它指向的对象
p->mem	等价于 (*p).mem
p.get()	返回 p 中保存的指针。要小心使用，若智能指针释放了其对象，返回的指针所指向的对象也就消失了
swap(p, q)	交换 p 和 q 中的指针
p.swap(q)	

https://blog.csdn.net/flowing_wind

如下表所示是shared_ptr特有的操作：

表 12.2: shared_ptr 独有的操作	
make_shared<T>(args)	返回一个 shared_ptr，指向一个动态分配的类型为 T 的对象。使用 args 初始化此对象
shared_ptr<T>p(q)	p 是 shared_ptr q 的拷贝；此操作会递增 q 中的计数器。q 中的指针必须能转换为 T*（参见 4.11.2 节，第 143 页）
p = q	p 和 q 都是 shared_ptr，所保存的指针必须能相互转换。此操作会递减 p 的引用计数，递增 q 的引用计数；若 p 的引用计数变为 0，则将其管理的原内存释放
p.unique()	若 p.use_count() 为 1，返回 true；否则返回 false
p.use_count()	返回与 p 共享对象的智能指针数量；可能很慢，主要用于调试

https://blog.csdn.net/flowing_wind

make_shared函数：

最安全的分配和使用动态内存的方法就是调用一个名为make_shared的标准库函数，此函数在动态内存中分配一个对象并初始化它，返回指向此对象的shared_ptr。头文件和share_ptr相同，在memory中必须指定想要创建对象的类型，定义格式见下面例子：

```
1 shared_ptr<int> p3 = make_shared<int>(42);
2 shared_ptr<string> p4 = make_shared<string>(10, '9');
3 shared_ptr<int> p5 = make_shared<int>();
```

make_shared用其参数来构造给定类型的对象，如果我们不传递任何参数，对象就会进行值初始化

shared_ptr的拷贝和赋值

当进行拷贝和赋值时，每个shared_ptr都会记录有多少个其他shared_ptr指向相同的对象。

```
1 auto p = make_shared<int>(42);
2 auto q(p);
```


我们可以认为每个shared_ptr都有一个关联的计数器，通常称其为引用计数，无论何时我们拷贝一个shared_ptr，计数器都会递增。当我们给shared_ptr赋予一个新值或是shared_ptr被销毁（例如一个局部的shared_ptr离开其作用域）时，计数器就会递减，一旦一个shared_ptr的计数器变为0,它就会自动释放自己所管理的对象。

```
1 auto r = make_shared<int>(42); //r指向的int只有一个引用者
2 r=q; //给r赋值，令它指向另一个地址
3     //递增q指向的对象的引用计数
4     //递减r原来指向的对象的引用计数
5     //r原来指向的对象已没有引用者，会自动释放
```

shared_ptr自动销毁所管理的对象

当指向一个对象的最后一个shared_ptr被销毁时，shared_ptr类会自动销毁此对象，它是通过另一个特殊的成员函数-析构函数完成销毁工作的，类似于构造函数，每个类都有一个析构函数。析构函数控制对象销毁时做什么操作。析构函数一般用来释放对象所分配的资源。**shared_ptr的析构函数会递减它所指向的对象的引用计数。如果引用计数变为0，shared_ptr的析构函数就会销毁对象，并释放它所占用的内存。**

shared_ptr还会自动释放相关联的内存

当动态对象不再被使用时，shared_ptr类还会自动释放动态对象，这一特性使得动态内存的使用变得非常容易。如果你将shared_ptr存放于一个容器中，而后不再需要全部元素，而只使用其中一部分，要记得用erase删除不再需要的那些元素。

使用了动态生存期的资源的类：

程序使用动态内存的原因：

- (1) 程序不知道自己需要使用多少对象
- (2) 程序不知道所需对象的准确类型
- (3) 程序需要在多个对象间共享数据

直接管理内存

C++定义了两个运算符来分配和释放动态内存，new和delete，使用这两个运算符非常容易出错。

使用new动态分配和初始化对象

在自由空间分配的内存是无名的，因此new无法为其分配的对象命名，而是返回一个指向该对象的指针。

```
1 int *pi = new int; //pi指向一个动态分配的、未初始化的无名对象
```

此new表达式在自由空间构造一个int型对象，并返回指向该对象的指针

默认情况下，动态分配的对象是默认初始化的，这意味着内置类型或组合类型的对象的值将是未定义的，而类类型对象将用默认构造函数进行初始化。

```
1 string *ps = new string; //初始化为空string
2 int *pi = new int; //pi指向一个未初始化的int
```

我们可以直接使用直接初始化方式来初始化一个动态分配一个动态分配的对象。我们可以使用传统的构造方式，在新标准下，也可以使用列表初始化

```
1 int *pi = new int(1024);
2 string *ps = new string(10, '9');
3 vector<int> *pv = new vector<int>{0,1,2,3,4,5,6,7,8,9};
```

也可以对动态分配的对象进行初始化，只需在类型名之后跟一对空括号即可；

动态分配的const对象

```
1  const int *pci = new const int(1024);
2  //分配并初始化一个const int
3  const string *pcs = new const string;
4  //分配并默认初始化一个const的空string
```

类似其他任何const对象，一个动态分配的const对象必须进行初始化。对于一个定义了默认构造函数的类类型，其const动态对象可以隐式初始化，而其他类型的对象就必须显式初始化。由于分配的对象就必须显式初始化。由于分配的对象是const的，new返回的指针就是一个指向const的指针。

内存耗尽：

虽然现代计算机通常都配备大容量内存，但是自由空间被耗尽的情况还是有可能发生。一旦一个程序用光了它所有可用的空间，new表达式就会失败。默认情况下，如果new不能分配所需的内存空间，他会抛出一个bad_alloc的异常，我们可以改变使用new的方式来阻止它抛出异常

```
1  //如果分配失败，new返回一个空指针
2  int *p1 = new int; //如果分配失败，new抛出std::bad_alloc
3  int *p2 = new (nothrow)int; //如果分配失败，new返回一个空指针
```

我们称这种形式的new为定位new,定位new表达式允许我们向new传递额外的参数，在例子中我们传给它一个由标准库定义的nothrow的对象，如果将nothrow传递给new，我们的意图是告诉它不要抛出异常。如果这种形式的new不能分配所需内存，它会返回一个空指针。bad_alloc和nothrow都在头文件new中。

释放动态内存

为了防止内存耗尽，在动态内存使用完之后，必须将其归还给系统，使用delete归还。

指针值和delete

我们传递给delete的指针必须指向动态内存，或者是一个空指针。释放一块并非new分配的内存或者将相同的指针释放多次，其行为是未定义的。即使delete后面跟的是指向静态分配的对象或者已经释放的空间，编译还是能够通过，实际上是错误的。

动态对象的生存周期直到被释放时为止

由shared_ptr管理的内存在最后一个shared_ptr销毁时会被自动释放，但是通过内置指针类型来管理的内存就不是这样了，内置类型指针管理的动态对象，直到被显式释放之前都是存在的，所以调用这必须记得释放内存。

使用new和delete管理动态内存常出现的问题：

- (1) 忘记delete内存
- (2) 使用已经释放的对象
- (3) 同一块内存释放两次

delete之后重置指针值

在delete之后，指针就变成了空悬指针，即指向一块曾经保存数据对象但现在已经无效的内存的地址

有一种方法可以避免悬空指针的问题：在指针即将要离开其作用域之前释放掉它所关联的内存

如果我们需要保留指针可以在delete之后将nullptr赋予指针，这样就清楚的指出指针不指向任何对象。

动态内存的一个基本问题是可能多个指针指向相同的内存

shared_ptr和new结合使用

如果我们不初始化一个智能指针，它就会被初始化成一个空指针，接受指针参数的智能指针是explicit的，因此我们不能将一个内置指针隐式转换为一个智能指针，必须直接初始化形式来初始化一个智能指针

```
1 shared_ptr<int> p1 = new int(1024); //错误：必须使用直接初始化形式
2 shared_ptr<int> p2(new int(1024)); //正确：使用了直接初始化形式
```

下表为定义和改变shared_ptr的其他方法：

shared_ptr<T> p(q)	p 管理内置指针 q 所指向的对象；q 必须指向 new 分配的内存，且能够转换为 T* 类型
shared_ptr<T> p(u)	p 从 unique_ptr u 那里接管了对象的所有权；将 u 置为空
shared_ptr<T> p(q, d)	p 接管了内置指针 q 所指向的对象的所有权。q 必须能转换为 T* 类型（参见 4.11.2 节，第 143 页）。p 将使用可调用对象 d（参见 10.3.2 节，第 346 页）来代替 delete

12.1 动态内存与智能指针

续表

shared_ptr<T> p(p2, d)	如表 12.2 所示，p 是 shared_ptr p2 的拷贝，唯一的区别是 p 将用可调用对象 d 来代替 delete
p.reset()	若 p 是唯一指向其对象的 shared_ptr，reset 会释放此对象。若传递了可选的参数内置指针 q，会令 p 指向 q，否则会将 p 置为空。若还传递了参数 d，将会调用 d 而不是 delete 来释放 q
p.reset(q)	
p.reset(q, d)	

https://blog.csdn.net/flowing_wind

不要混合使用普通指针和智能指针

如果混合使用的话，智能指针自动释放之后，普通指针有时就会变成悬空指针，当将一个shared_ptr绑定到一个普通指针时，我们就将内存的管理责任交给了这个shared_ptr。一旦这样做了，我们就不应该再使用内置指针来访问shared_ptr所指向的内存了。

也不要使用get初始化另一个智能指针或为智能指针赋值

```
1 shared_ptr<int> p(new int(42)); //引用计数为1
2 int *q = p.get(); //正确：但使用q时要注意，不要让它管理的指针被释放
3 {
4     //新程序块
5     //未定义：两个独立的share_ptr指向相同的内存
6     shared_ptr(q);
7
8 } //程序块结束，q被销毁，它指向的内存被释放
9 int foo = *p; //未定义，p指向的内存已经被释放了
```

p和q指向相同的一块内部才能，由于是相互独立创建，因此各自的引用计数都是1，当q所在的程序块结束时，q被销毁，这会导致q指向的内存被释放，p这时候就变成一个空悬指针，再次使用时，将发生未定义的行为，当p被销毁时，这块空间会被二次delete

其他shared_ptr操作

可以使用reset来将一个新的指针赋予一个shared_ptr:

```
1 p = new int(1024); //错误: 不能将一个指针赋予shared_ptr
2 p.reset(new int(1024)); //正确. p指向一个新对象
```

与赋值类似，reset会更新引用计数，如果需要的话，会释放p的对象。reset成员经常和unique一起使用，来控制多个shared_ptr共享的对象。在改变底层对象之前，我们检查自己是否是当前对象仅有的用户。如果不是，在改变之前要制作一份新的拷贝：

```
1 if(!p.unique())
2 p.reset(new string(*p)); //我们不是唯一用户，分配新的拷贝
3 *p+=newVal; //现在我们知道自己是唯一的用户，可以改变对象的值
```

智能指针和异常

如果使用智能指针，即使程序块过早结束，智能指针也能确保在内存不再需要时将其释放，sp是一个shared_ptr,因此sp销毁时会检测引用计数，当发生异常时，我们直接管理的内存是不会自动释放的。如果使用内置指针管理内存，且在new之后在对应的delete之前发生了异常，则内存不会被释放。

使用我们自己的释放操作

默认情况下，shared_ptr假定他们指向的是动态内存，因此当一个shared_ptr被销毁时，会自动执行delete操作，为了用shared_ptr来管理一个connection，我们必须首先必须定义一个函数来代替delete。这个删除器函数必须能够完成对shared_ptr中保存的指针进行释放的操作。

智能指针陷阱：

- (1) 不使用相同的内置指针值初始化（或reset）多个智能指针。
- (2) 不delete get()返回的指针
- (3) 不使用get()初始化或reset另一个智能指针
- (4) 如果你使用get()返回的指针，记住当最后一个对应的智能指针销毁后，你的指针就变为无效了
- (5) 如果你使用智能指针管理的资源不是new分配的内存，记住传递给它一个删除器

unique_ptr

某个时刻只能有一个unique_ptr指向一个给定对象，由于一个unique_ptr拥有它指向的对象，因此unique_ptr不支持普通的拷贝或赋值操作。

下表是unique的操作：

表 12.4: unique_ptr 操作（另参见表 12.1，第 401 页）	
unique_ptr<T> u1	空 unique_ptr，可以指向类型为 T 的对象。u1 会使用 delete 来释放它的指针；u2 会使用一个类型为 D 的可调用对象来释放它的指针
unique_ptr<T, D> u2	
unique_ptr<T, D> u(d)	空 unique_ptr，指向类型为 T 的对象，用类型为 D 的对象 d 代替 delete
u = nullptr	释放 u 指向的对象，将 u 置为空
u.release()	u 放弃对指针的控制权，返回指针，并将 u 置为空
u.reset()	释放 u 指向的对象
u.reset(q)	如果提供了内置指针 q，令 u 指向这个对象；否则将 u 置为空
u.reset(nullptr)	

虽然我们不能拷贝或者赋值unique_ptr，但是可以通过调用release或reset将指针所有权从一个（非const）unique_ptr转移给另一个unique_ptr

```
1 //将所有权从p1（指向string Stegosaurus）转移给p2
2 unique_ptr<string> p2(p1.release()); //release将p1置为空
3 unique_ptr<string> p3(new string("Trex"));
4 //将所有权从p3转移到p2
5 p2.reset(p3.release()); //reset释放了p2原来指向的内存
```

release成员返回unique_ptr当前保存的指针并将其置为空。因此，p2被初始化为p1原来保存的指针，而p1被置为空。

reset成员接受一个可选的指针参数，令unique_ptr重新指向给定的指针。

调用release会切断unique_ptr和它原来管理的对象间的联系。release返回的指针通常被用来初始化另一个智能指针或给另一个智能指针赋值。

不能拷贝unique_ptr有一个例外：我们可以拷贝或赋值一个将要被销毁的unique_ptr。最常见的例子是从函数返回一个unique_ptr。

```
1 unique_ptr<int> clone(int p)
2 {
3     //正确：从int*创建一个unique_ptr<int>
4     return unique_ptr<int>(new int(p));
5 }
```

还可以返回一个局部对象的拷贝：

```
1 unique_ptr<int> clone(int p)
2 {
3     unique_ptr<int> ret(new int(p));
4     return ret;
5 }
```


向后兼容: auto_ptr

标准库的较早版本包含了一个名为auto_ptr的类, 它具有uniqued_ptr的部分特性, 但不是全部。

用unique_ptr传递删除器

unique_ptr默认使用delete释放它指向的对象, 我们可以重载一个unique_ptr中默认的删除器

我们必须在尖括号中unique_ptr指向类型之后提供删除器类型。在创建或reset一个这种unique_ptr类型的对象时, 必须提供一个指定类型的可调用对象删除器。

weak_ptr

weak_ptr是一种不控制所指向对象生存期的智能指针, 它指向一个由shared_ptr管理的对象, 将一个weak_ptr绑定到一个shared_ptr不会改变shared_ptr的引用计数。一旦最后一个指向对象的shared_ptr被销毁, 对象就会被释放, 即使有weak_ptr指向对象, 对象还是会被释放。

weak_ptr的操作

表 12.5: weak_ptr	
weak_ptr<T> w	空 weak_ptr 可以指向类型为 T 的对象
weak_ptr<T> w(sp)	与 shared_ptr sp 指向相同对象的 weak_ptr。T 必须能转换为 sp 指向的类型
w = p	p 可以是一个 shared_ptr 或一个 weak_ptr。赋值后 w 与 p 共享对象
w.reset()	将 w 置为空
w.use_count()	与 w 共享对象的 shared_ptr 的数量
w.expired()	若 w.use_count() 为 0, 返回 true, 否则返回 false
w.lock()	如果 expired 为 true, 返回一个空 shared_ptr; 否则返回一个指向 w 的对象的 shared_ptr

由于对象可能不存在, 我们不能使用weak_ptr直接访问对象, 而必须调用lock, 此函数检查weak_ptr指向的对象是否存在。如果存在, lock返回一个指向共享对象的shared_ptr, 如果不存在, lock将返回一个空指针

scoped_ptr

scoped和weak_ptr的区别就是, 给出了拷贝和赋值操作的声明并没有给出具体实现, 并且将这两个操作定义成私有的, 这样就保证scoped_ptr不能使用拷贝来构造新的对象也不能执行赋值操作, 更加安全, 但有了"++""--"以及"++""--"以及"*" ">" 这些操作, 比weak_ptr能实现更多功能。

16、shared_ptr技术与陷阱

①意外延长对象的生命期

- shared_ptr是强引用, 只要有一个指向x对象的shared_ptr存在, 该对象就不会析构
- 而shared_ptr又是允许拷贝构造和赋值的, 如果不小心遗留了一个拷贝, 那么对象就永世长存了。例如:
 - ①前面提到如果把“五”的代码中把observers_的类型改为vector<shared_ptr>, 那么除非手动调用unregister(), 否则Observer对象永远不会析构。即便它的析构函数会调用unregister(), 但是不去unregister()就不会调用Observer的析构(函数。这也是Java内存泄漏的常见原因。)

- ②另外一个出错的可能是`boost::bind`，因为`std::bind`会把实参拷贝一份，如果参数是个`shared_ptr`，那么对象的生命期就不会短于`std::function`对象：

```
1 #include <functional>;
2
3 class Foo
4 {
5 public:
6 void doit();
7 };
8
9 shared_ptr<Foo> pFoo(new Foo);
10 std::function<void()> func = std::bind(&Foo::doit, pFoo);
```

- 这里`func`对象持有了`shared_ptr`的一份拷贝，有可能会在不经意间延长倒数第二行创建的`Foo`对象的生命期

②函数参数

- 因为要修改引用计数（而且拷贝的时候通常要加锁），`shared_ptr`的拷贝开销比拷贝原始指针要高，但是需要拷贝的时候并不多。多数情况下它可以以`const reference`方式传递，一个线程只需要在最外层函数有一个实体对象，之后都可以用`const reference`来使用这个`shared_ptr`。例如有几个函数都要用到`Foo`对象：

```
1 void save(const std::shared_ptr<Foo>& pFoo); //pass by const reference
2 void validateAccount(const Foo& foo);
3
4 bool validate(const std::shared_ptr<Foo>& pFoo) //pass by const reference
5 {
6     validateAccount(*pFoo);
7 }
```

- 那么在通常情况下，我们可以传常引用：

```
1 void onMessage(const string& msg)
2 {
3     std::tr1::shared_ptr<Foo> pFoo(new Foo(msg)); //只要在最外层持有一个实体，安全就不是问题
4     if (validate(pFoo)) { //没有拷贝pFoo
5         save(pFoo); //没有拷贝pFoo
6     }
7 }
```

- 遵照这个规则：
 - 基本上不会遇到反复拷贝`shared_ptr`导致的性能问题
 - 另外由于`pFoo`是栈上对象，不可能被别的线程看到，那么读取始终是线程安全的

③析构动作在创建时被捕获

- 这是一个非常有用的特性，这意味着：
 - 虚析构不再是必需的

- `shared_ptr`可以持有任何对象，而且能安全地释放
- `shared_ptr`对象可以安全地跨越模块边界，比如从DLL里返回，而不会造成从模块A分配的内存存在模块B里被释放这种错误
- **二进制兼容性，即便Foo对象的大小变了**，那么旧的客户代码仍然可以使用新的动态库，而无须重新编译。前提是Foo的头文件中不出现访问对象的成员的inline函数，并且Foo对象的由动态库中的Factory构造，返回其`shared_ptr`
- **析构动作可以定制（通过`shared_ptr`的参数2）**
- 最后这个特性的实现比较巧妙，因为`shared_ptr`只有一个模板参数，而“析构行为”可以是函数指针、仿函数（functor）或者其他什么东西。这是泛型编程和面向对象编程的一次完美结合（这个技术在后面的对象池中还会用到）

④析构所在的线程

- 析构所在的线程对象的析构是同步的，当最后一个指向x的`shared_ptr`离开其作用域的时候，x会同时在同一个线程析构。这个线程不一定是对象诞生的线程
- 这个特性是把双刃剑：
 - **缺点**：如果对象的析构比较耗时，那么可能会拖慢关键线程的速度（如果最后一个`shared_ptr`引发的析构发生在关键线程）
 - **优点**：同时，我们可以用一个单独的线程来专门做析构，通过一个`BlockingQueue<shared_ptr>`把对象的析构都转移到那个专用线程，从而解放关键线程

⑤线程的RAII handle

- 我认为RAII（资源获取即初始化）是C++语言区别于其他所有编程语言的最重要的特性，一个不懂RAII的C++程序员不是一个合格的C++程序员
- 初学C++的教条是“new和delete要配对，new了之后要记着delete”
- 如果使用RAII（参阅Effective C++：
https://blog.csdn.net/qq_41453285/article/details/104219176），要改成“每一个明确的资源配置动作（例如new）都应该在单一语句中执行，并在该语句中立刻将配置获得的资源交给handle对象（如`shared_ptr`），程序中一般不出现delete”
- **`shared_ptr`是管理共享资源的利器，需要注意避免循环引用**，通常的做法是owner持有指向child的`shared_ptr`，child持有指向owner的`weak_ptr`

19、说说 C++ 中智能指针和指针的区别是什么？

1. 智能指针 如果在程序中使用new从堆（自由存储区）分配内存，等到不需要时，应使用delete将其释放。C++引用了智能指针`auto_ptr`，以帮助自动完成这个过程。随后的编程体验（尤其是使用STL）表明，需要有更精致的机制。基于程序员的编程体验和BOOST库提供的解决方案，C++11摒弃了`auto_ptr`，并新增了三种智能指针：**`unique_ptr`、`shared_ptr`和`weak_ptr`**。所有新增的智能指针都能与STL容器和移动语义协同工作。
2. 指针 C语言规定所有变量在使用前必须先定义，指定其类型，并按此分配内存单元。指针变量不同于整型变量和其他类型的变量，它是专门用来存放地址的，所以必须将它定义为“指针类型”。
3. 智能指针和普通指针的区别 **智能指针和普通指针的区别在于智能指针实际上是对普通指针加了一层封装机制，区别是它负责自动释放所指的物体，这样的一层封装机制的目的是为了使得智能指针可以方便的管理一个对象的生命期。**

22、weak_ptr 能不能知道对象计数为 0，为什么？

不能。weak_ptr是一种不控制对象生命周期的智能指针，它指向一个shared_ptr管理的对象。进行该对象管理的是那个引用的shared_ptr。weak_ptr只是提供了对管理对象的一个访问手段。weak_ptr设计的目的只是为了配合shared_ptr而引入的一种智能指针，配合shared_ptr工作，它只可以从一个shared_ptr或者另一个weak_ptr对象构造，弱引用，它的构造和析构不会引起计数的增加或减少。

24、share_ptr 怎么知道跟它共享对象的指针释放了

多个shared_ptr对象可以同时托管一个指针，系统会维护一个托管计数。当无shared_ptr托管该指针时，delete该指针。

26、shared_ptr线程安全

多线程环境下，调用不同shared_ptr实例的成员函数是不需要额外的同步手段的，即使这些shared_ptr拥有的是同样的对象。但是如果多线程访问（有写操作）同一个shared_ptr，则需要同步，否则就会有race condition发生。也可以使用shared_ptr overloads of atomic functions来防止race condition的发生。

多个线程同时读同一个shared_ptr对象是线程安全的，但是如果是多个线程对同一个shared_ptr对象进行读和写，则需要加锁。

多线程读写shared_ptr所指向的同一个对象，不管是相同的shared_ptr对象，还是不同的shared_ptr对象，也需要加锁保护。例子如下：

```
1 //程序实例
2 shared_ptr<long> global_instance = make_shared<long>(0);
3 std::mutex g_i_mutex;
4
5 void thread_fcn()
6 {
7     //std::lock_guard<std::mutex> lock(g_i_mutex);
8 }
```

```

9      //shared_ptr<long> local = global_instance;
10
11      for(int i = 0; i < 100000000; i++)
12      {
13          *global_instance = *global_instance + 1;
14          /*local = *local + 1;
15      }
16  }
17
18  int main(int argc, char** argv)
19  {
20      thread thread1(thread_fcn);
21      thread thread2(thread_fcn);
22
23      thread1.join();
24      thread2.join();
25
26      cout << "*global_instance is " << *global_instance << endl;
27
28      return 0;
29  }

```

在线程函数thread_fcn的for循环中，2个线程同时对global_instance进行加1的操作。这就是典型的非线程安全的场景，最后的结果是未定的，运行结果如下：

```
1 | *global_instance is 197240539
```

如果使用的是每个线程的局部shared_ptr对象local，因为这些local指向相同的对象，因此结果也是未定的，运行结果如下：

```
1 | *global_instance is 160285803
```

因此，这种情况下必须加锁，将thread_fcn中的第一行代码的注释去掉之后，不管是使用 global_instance，还是使用local，得到的结果都是：

```
1 | *global_instance is 200000000
```

28、请你回答一下智能指针有没有内存泄露的情况

智能指针有内存泄露的情况发生。

1. 智能指针发生内存泄露的情况 当两个对象同时使用一个shared_ptr成员变量指向对方，会造成**循环引用，使引用计数失效，从而导致内存泄露**。
2. 智能指针的内存泄漏如何解决？为了解决循环引用导致的内存泄漏，引入了弱指针weak_ptr，**weak_ptr的构造函数不会修改引用计数的值，从而不会对对象的内存进行管理**，其类似一个普通指针，但是不会指向引用计数的共享内存，但是**可以检测到所管理的对象是否已经被释放，从而避免非法访问**。

