|  |  |
| --- | --- |
| 1. 智能指针和异常 2. 智能指针和哑类 3. 使用我们自己的操作 | 删除器  Void f(){  Shared\_ptr<int> sp1(new int(42));  // 抛出异常，并未在f中捕获，则中断下面代码由上查找  }  Void f(){  Int \*ip = new int(42);  // 抛出异常，并未在f中捕获，则中断下面代码由上查找  Delete ip;//未执行  }  Void f(){  Connection c ;  //若没有显式调用销毁代码，无法关闭c了  }  void deletefunc(connection\*p)  {  Disconnect(\*p);  }  Void f(destination &d){  Connection c = connect(&d);  Shared\_ptr<connection> p(&c,end\_connection);  } |
| 1. 智能指针和异常 2. 若程序块过早结束（正常处理或发生异常不在本程序块捕获，向上查找），局部对象会被销毁，但是若是动态分配的对象则需要看什么指针类型 3. 若自己管理动态内存，在最后面进行delete，delete代码不执行，指向的内存将不会释放，若智能指针函数结束时还是会执行 4. 智能指针和哑类 5. 很多类都定义了析构函数清理对象使用的资源，不是所有类都有这种良好的定义，特别是为c和c++两种语言设计的类，通常需要显式的释放。就是例如内置类型对象离开作用域自动销毁，但是这种需要显式释放的对象没有析构函数，则需要手动删除，就是delete 指针 一样要显式调用关闭的代码。 6. 但是如果我们忘记显式调用则无法关闭和释放了。 7. 使用我们自己的操作 8. 可以使用shared\_ptr来正确释放。但是shared\_ptr默认是指向动态内存，所以需要传递指针类型。需定义一个函数来作为删除器。必须能够完成对shared\_Ptr中保存的指针进行释放的操作 9. 在创建shared\_ptr时指定这个删除器方法就行。就是调用具体的方法来释放 10. 可用lambda也可以用函数自定义删除   注意：智能指针陷阱   1. 不适用相同的内置指针初始化多个智能指针 2. 不delete get() 返回的指针 3. 不适用get() 初始化或reset另一个智能指针 4. 适用get（）返回的指针，记住最后一个对象的智能指针销毁后，指针变为无效 5. 如果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 | |