|  |  |
| --- | --- |
| 1. 交换操作 2. 编写我们自己的swap函数 3. Swap函数应该调用swap，而不是std::swap 4. 在赋值运算符中使用swap | HasPtr temp = v1;  V1 = v2;  V2 = temp;  String \*temp = v1.ps;  V1.ps = v2.ps;  V2.ps = temp;  Class HasPtr{  Friend void swap(HasPtr&,HasPtr&);  }  Inline  Void swap(Hasptr &lhs,Hasptr& rhs)  {  Using std::swap;  Swap(lhs.ps,rhs.ps);  Swap(lhs.i,rhs.i);  }  拷贝并交换  HasPtr& HasPtr::operator=(HasPtr rhs)  {  Swap(\*this,rhs);  Return \*this;  } |
| 1. 交换操作 2. 除了定义拷贝控制成员，管理资源的类通常还定义一个名为swap的函数 3. 对于那些与重排元素顺序的算法一起使用的类，定义swap是重要的。 4. 如果类自己定义了swap就使用类自定义版本，否则，算法将使用标准定义的swap 5. 标准库的swap交换两个对象需要进行一次拷贝和两次赋值。 6. 但是这些花销是可以不需要的。我们希望swap交换指针，而不是分配string的新副本。则可以自定义swap交换指针 7. 编写我们自己的swap函数 8. 定义swap为friend，需声明inline优化代码。   Note：与拷贝控制成员不同，swap并不是必要的。但是对于分配了资源的类，定义swap可能是一个很重要的优化手段   1. Swap函数应该调用swap，而不是std::swap 2. 内置类型是不需要特定版本的swap的(不需要自定义)，对内置类型则需要使用标准库的swap 3. 若是动态分配资源的成员，我们需要自定义swap，而不是用标准库swap，因为它会进行一次拷贝和两次赋值（操作内容），我们自定义swap是操作指针 4. 为了匹配上面两点，不能直接std::swap(I,j);//交换，需要   Using std::swap  Swpa(lhs.h,rhs,h);//若是指针定义了这个版本使用我们定义的，优先于标准库。  Swap(lhs.i,rhs.i);//若是内置类型并且没有自定义这个版本则使用标准库  匹配过程：   1. using声明将std::swap作用于swap作用域内，说不会隐藏外层自定义的swap，并且外部swap优先匹配标准库的swap   不会隐藏外层自定义的swap：P210，我觉得会隐藏，需要把using声明在外面才可，这样才不会隐藏。或者不是用using声明，而是使用using指示将作用域提升到全局就不会覆盖隐藏外部的了。  外部swap优先匹配标准库的swap：P616，因为非模板类型比模板类型更优先。   1. 在赋值运算符中使用swap 2. 这个版本的参数不是引用，以传值方式传递给了赋值运算符。Rhs是右侧运算对象的一个副本，会使用拷贝构造函数，rhs计数+1 3. Swap完成后，\*this指向新分配的string（参数的string）和计数器，参数指向原来的\*this 4. 结束返回\*this，它指向右侧运算对象的指针，完成了右侧运算对象+1的工作   Rhs作用域结束，销毁，使原来的左侧运算对象计数-1，析构左侧运算对象，完成左侧运算对象-1的工作   1. 有趣的地方是自动处理了自赋值情况是安全的   如：  拷贝副本后，计数为2，交换后，this计数为2，副本计数也为2，结束作用域副本计数为1，this也为1，rhs也为1   1. 将左侧与右侧的一个副本进行交换 2. 保证异常安全的方法也与原来的赋值运算符实现一样的 3. 自动是异常安全的，并且能正确处理自赋值 | |