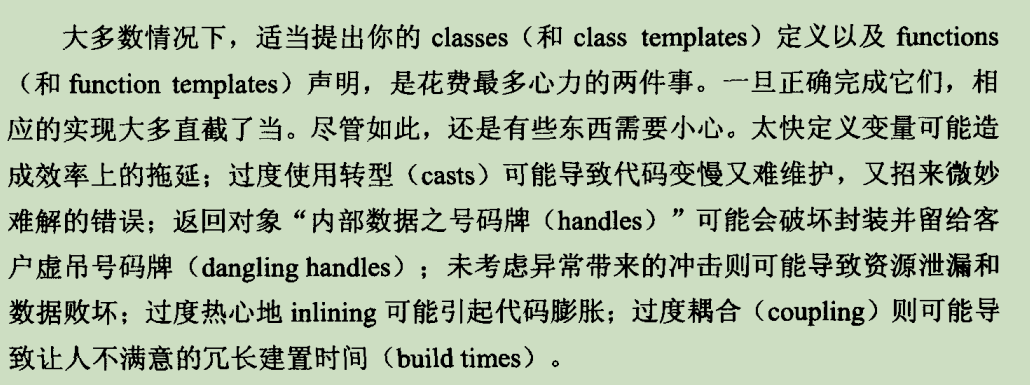


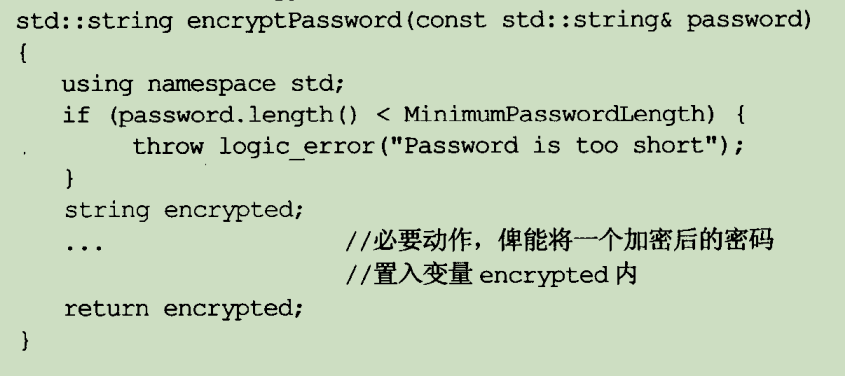
**解决问题**



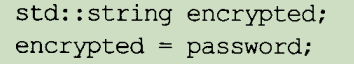
**条款27**

如果if被调用报错，那么encrypted的定义就没有任何意义，构造和析构函数时间浪费

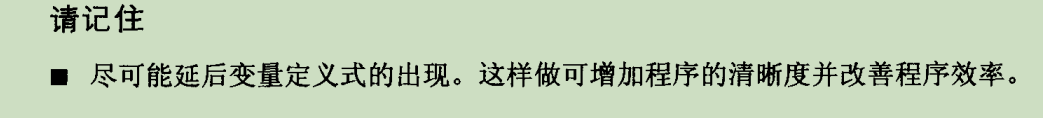




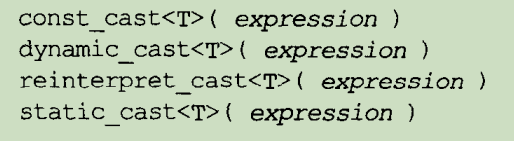
后面直接构造，前者无用构造再赋值，避免

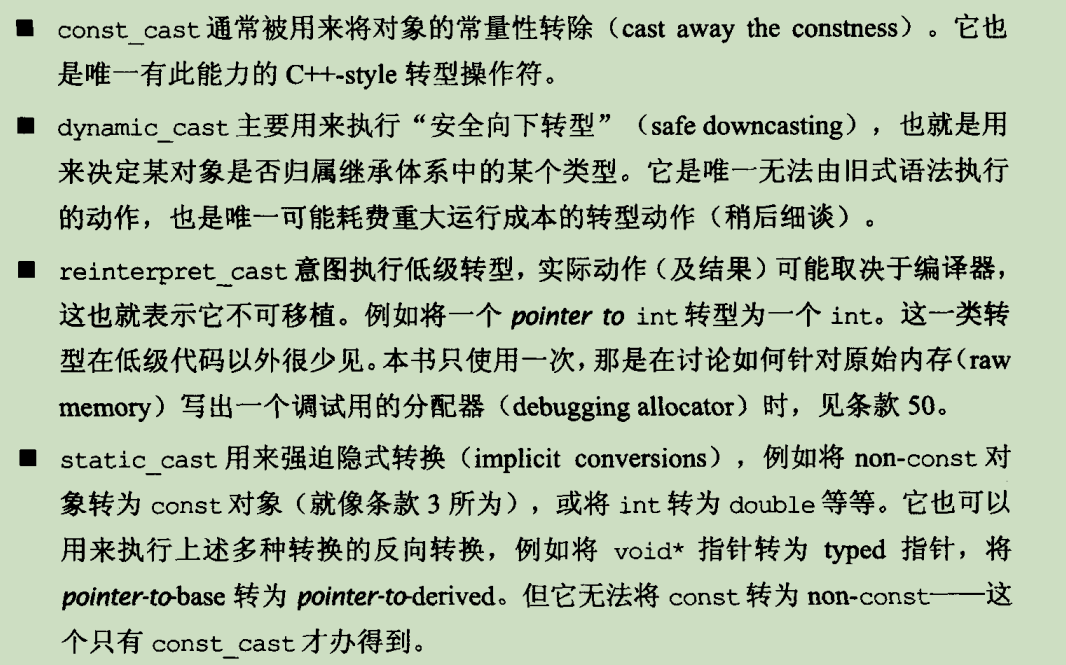


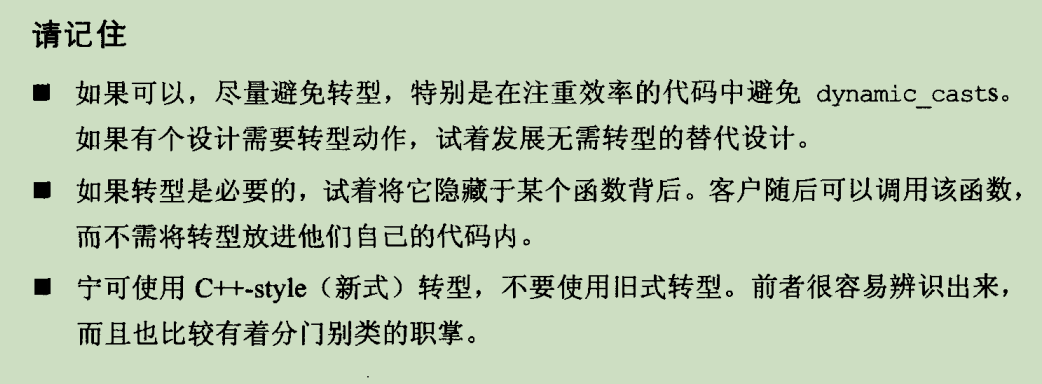
clipboard.png



条款27







**条款28**

**不要返回一个私有成员的引用**

为了提高效率，我们返会reference代替value

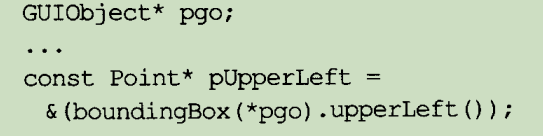
我们返回一个私有成员的指针

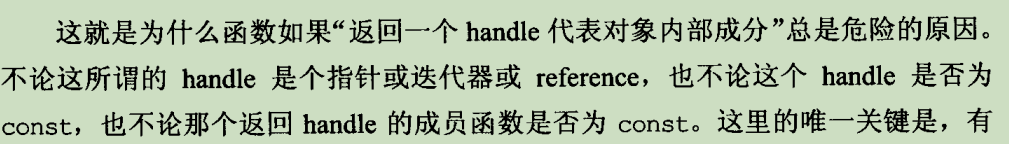
clipboard.png

同样这个私有成员暴露，失去了私有成员的封装意义。我们可以采用添加一个const，让他不可更改

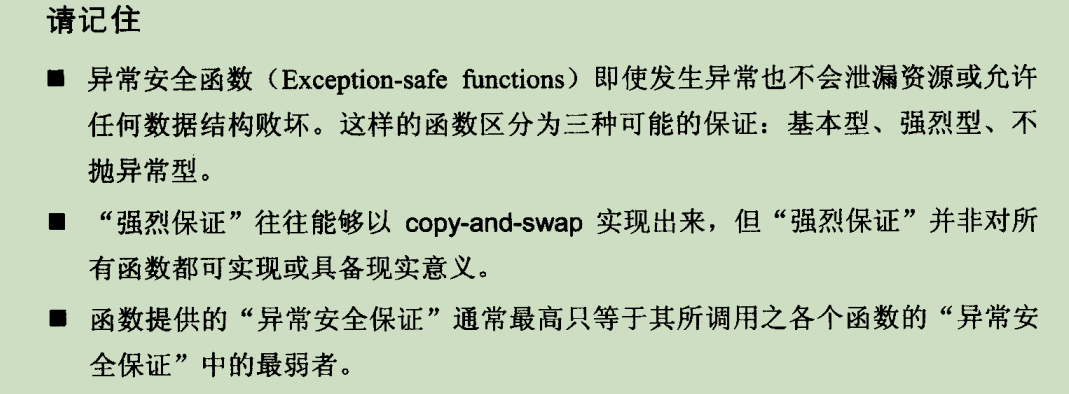
clipboard.png

这种方式，我们这个bouningBox这个函数创建一个临时对象，返回前面的指针，我们用一个新指针a指向这个指针，在这条语句结束以后，对象消灭，指向了无意义指针。





**条款29**

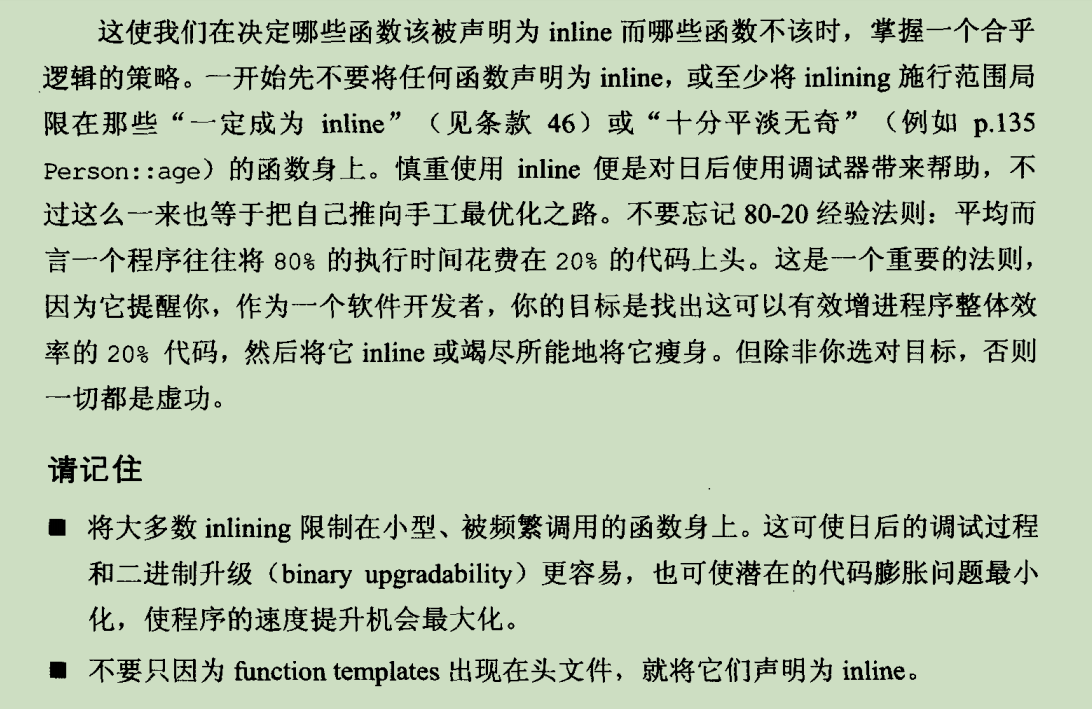


这一条读的不是很懂

为了保证安全，也就是强烈保证函数，我们通过copy-and-swap来实现，这个的思路就是，我们书写的函数某一处出现内存泄漏等等，我们希望回到计算机读取之前这一处的状态，我们把原对象，复制到一个副本中，对副本进行更改，如果这一处出现问题，也就是对副本更改出现问题（例如副本失败为无效 的指针），那我们swap的交换就不会成功，也就是回到之前的状态

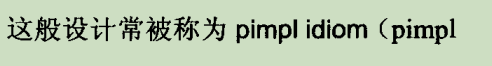
**条款30**

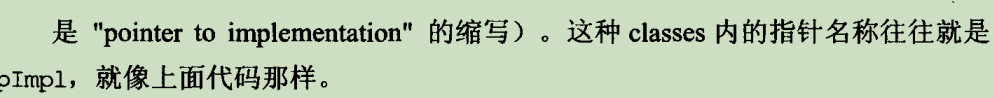
inline的内涵在于 编译器直接在调用他的位置替换成这个inline函数

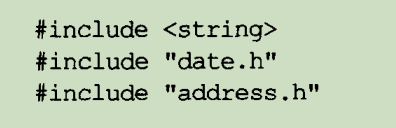


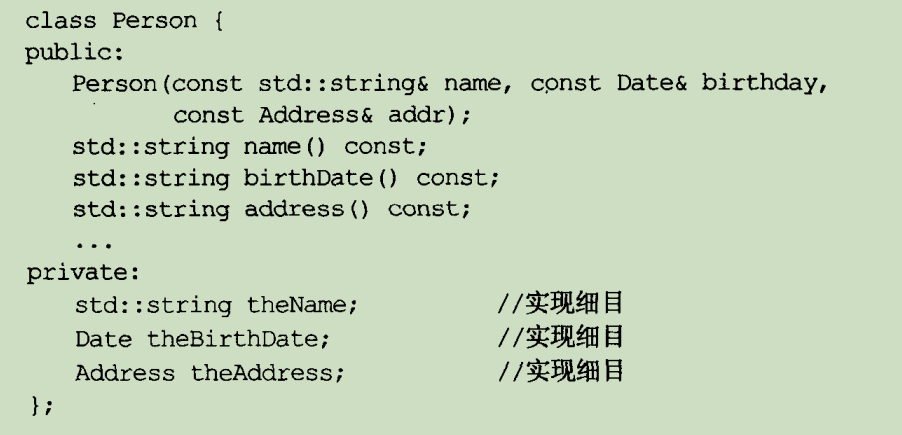
**条款31**

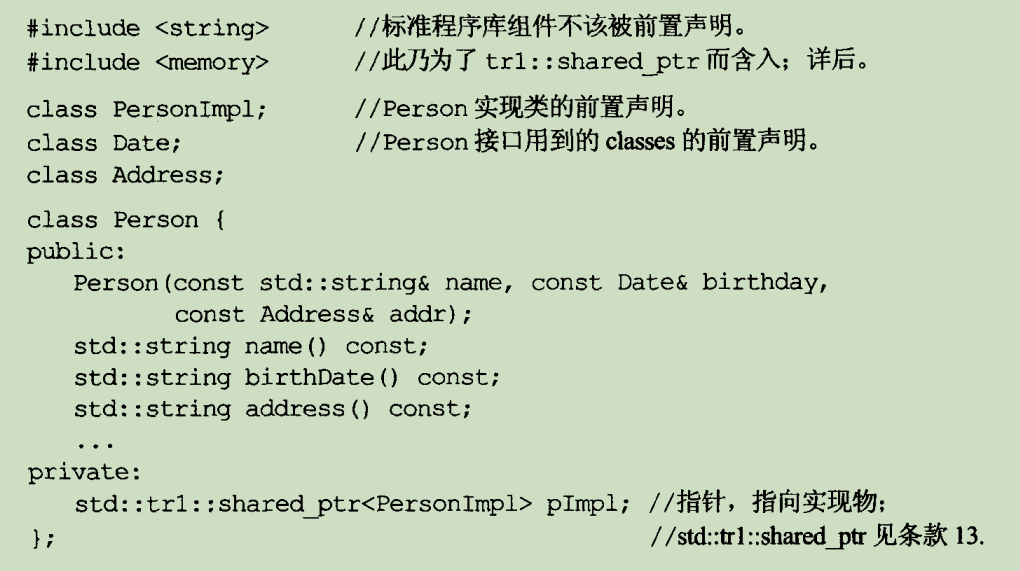
下面两种图片不同的写法，下面的图片把接口和对象真正的分开了，如果我们是person的用户，如果我们自己改变了date的数据，那么由于我们写的是#include“data.h"的定义式，这样我们的person编译时，他会看data数据，所以需要重新编译，如果是 class data；就是声明式，person的编译不会去看他细节，这个pImpl的写法，更是将数据隐藏，personImpl的声明，隐藏了很多，这种写法就大大降低了文件之间的依赖性，person用户编译的时间大大缩小。

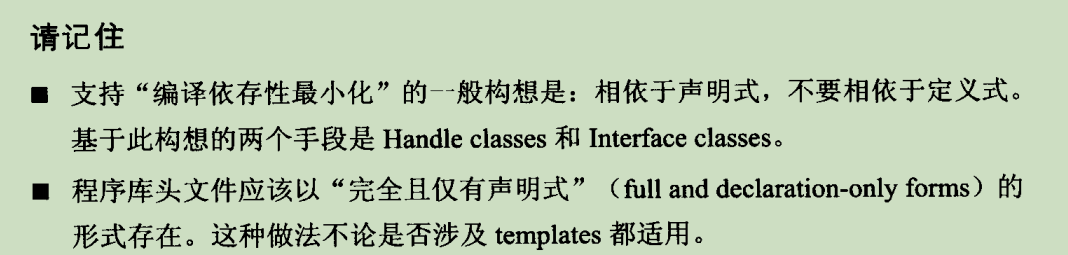












**条款32**