Exceptional C++

原著：Herb Sutter
Kingofark （4～19、26～29、35、36、38～44、47）
侯捷 （前言、序、译序、1、20～25、后记）
WQ （2/3、30～34、37、46）
Vcmfc （45）

[Herb Sutter 对 GotW 所有条目的申明]：这是 GotW 的问题和解答发表在 Usenet 上时的原作。查阅《Exceptional C++》（Addison-Wesley，2000）以得知 GotW #1-30 的最新解答。与最初的 GotW 相比，书上的解答作了的修正和扩充，并以 ANSI/ISO 的 C++标准审阅过。

[kingofark 对其所译条目的声明]：本文内容取自 www.gotw.ca 网站上的 Guru of the Week 栏目，其著作权归原著者本人所有。译者 kingofark 在未经原著者本人同意的情况下翻译本文。本翻译内容仅供自学和参考用，请所有阅读过本文的人不要擅自转载、传播本翻译内容；下载本翻译内容的人请在阅读浏览后，立即删除其备份。译者 kingofark 对违反上述两条原则的人不负任何责任。特此声明。

[WQ 对本文的申明]：由于《Exceptional C++》迟迟不见出版，遂作了这个以 GotW 估推的版本，虽不中，亦不远矣，聊解久盼之苦。传播本文内容时，请务必保留 kingofark 的声明。

1. 前言（Herb Sutter）

Exceptional C++ 以实例方式告诉你如何进行坚实的软体工程。本书涵盖 Guru of the Week（简称 GotW）前 30 个条款的扩充。GotW 是广受欢迎的网际网路 C++特别节目，是独立性极高的一系列 C++ 工程问题和解答，以实例说明特定的设计技术和编程技术。

本书并非随随便便挑拣一些编程难题；本书主要是做为一个导引，引导读者进入真实世界中的 C++ 企业级软体设计。本书采用「问题/解答」格式，这是我所知道最有效的教育法。把观念藏在问题之中，把道理藏在准则之中。虽然这些条款涵盖了许多主题，你会发现其中有些循环出现，专注於企业级的开发议题上，特别是异常发生时的安全性（exception safety）、健全的类别和模组设计、恰当的最佳化、可携而符合标准的码。

我希望你能够在这些题材中找到对你日常工作有益的东西。我也希望你能至少发现一些极好的思想和精致的技术，偶尔地，当你阅读到那些篇幅，突然发出一句『啊哈』。毕竟，谁说软体工程的生活呆滞、晦暗、无光采可言呢？

如何阅读本书

我假设你已经通晓基本的 C++。如果不是这样，请先从一本好的 C++ 导入性及概念性书籍开始（经典大部头作品如 Bjarne Stroustrup 的 The C++ Programming Language 第三版 1，Stan Lippman 和 Josée Lajoie 合著的 C++ Primer 第三版 2，都是很好的选择）。然後，请挑选一本教你程式风格的书籍，例如 Scott Meyers 的经典作品 Effective C++ 上下册（我发现这套书有 CD 版本，可使用浏览器阅读，十分方便好用）[注 3]

本书的每一个条款都以问题呈现，搭配一个导入性表头，看起来像这样：

```
条款## 题目主旨困难度：X
```

其中的主旨和难度分级（通常从 3 到 9? 不等，满分 10）给你提示，告诉你面对的是何方神圣。注意，难度分等是我自己的主观认定，我想像大部份人对各个问题的反应。你也有可能发现一个难度 7 的问题对你而言竟然比难度 5 的问题简单。当然，当你看到一个 9? 的怪物迎面而来，最好还是做好心理准备。

你不需要依序阅读各章节和各问题，不过有些是系列文章，你会看到「之一」、「之二」等等，甚至高达「之十」。这些系列文章最好视为一个群组。

怎么会有这本书：GotW 和 PeerDirect

C++ Guru of the Week 由来已久。GotW 最初诞生於 1996 年末，为我们（PeerDirect 公司）自己的开发小组提供一些有趣的挑战和无间的教育。我写它以便提供一个愉快的学习工具，涵盖的主题包括 inheritance 和 exception safety 的适当运用。後来我也利用它做为一个窗口，被我的团队了解 C++ 标准会议所做的各项改变。从那时候起，GotW 成为网际网路讨论群组 comp.lang.c++.moderated 上一个常态性的 C++ 公开节目，在那里你可以找到後续的问题和解答（以及许多有趣的讨论）。在 PeerDirect 公司里，把 C++ 要得好是很重要的，其理由和你的公司重视 C++没有两样— 虽然也许目标不同。我们主要制造系统软体，用於分散式资料库和资料库的复制— 专注於企业议题：稳定、安全、可携、高效…。我们写的软体必须能够移植到各种编译器和各个作业平台上。它必须在面对资料库交易出现死结或通讯中断或异常时，安全而稳健。它被客户用来管理智慧卡、自动机、PalmOS 和 WinCE 装置内的小型资料库，以及 Windows NT, Linux, Solaris 伺服器、Web 伺服器和资料大厂的大块头 Oracle 後端机器的大型资料库— 以相同的软体、相同的可信度、相同的程式码。现在，当我们爬行於 50 万行无注解的码，面临了可携性和可信度的重大挑战。

对於过去数年来在网际网路阅读 Guru of the Week 的人，我有些话要说：

- 感谢你们所展现的兴趣、支持、电子邮件、赞美、修正、注解、批评、发问—特别是你们要求 GotW 组装为书籍型式。本书就是我的回应，希望你喜欢。
- 本书包含的东西比你在网际网路上看到的更多得多。

Exceptional C++ 不是一本剪贴簿，它并不是撷取虚拟空间中的 GotW 文章再剪剪贴贴而已。书中所有的问题和解答都经过校订并重做，例如条款 8~17 所讨论的 exception safety，最初只是 GotW 上单独一篇文章，现在变成了由十篇文章组成的迷你系列。每一个问题和解答都以新的 C++ 标准规格重新检阅过。

所以，如果你一直是 GotW 的常态读者，本书对你而言仍有新东西可看。面对所有忠实读者，再次致上我的感谢，希望这些东西能够帮助你继续磨练并扩充你的软体工程素养和 C++ 编程技巧。

致谢

中文版略

Herb Sutter
June 1999

注 1： Stroustrup B. The C++ Programming Language, Third Edition (Addison Wesley Longman,1997).
注 2：Lippman S. and Lajoie J. C++ Primer, Third Edition (Addison Wesley Longman, 1998).
注 3： Meyers S. Effective C++ CD: 85 Specific Ways to Improve Your Programs and Designs(Addison Wesley Longman, 1999).此产品在 http://www.meyerscd.awl.com 有线上展示。

2. 序（Scott Meyers）

这是一本值得注意的书，不过，直到我将整本书大略读完之後，我才真正了解到它有多么值得注意。这恐怕是第一本写给已经完全懂 C++ 的人看的书。从语言的特性到标准程式库内的组件，再到程式编写技术，本书在不同的主题之间跳跃，总是使你稍稍失去平衡，总是使你必须付出全然的注意力。就像真正的 C++ 程式一样。Class 的设计无意间遇上虚拟函式的行为、iterator 常用手法对上名称搜寻规则、赋值运算子（assignment operators）冲击异常安全性（exception safety），编译依存关系与 templates 纠葛纵横。就像它们在真实程式中的作为一样。这些东西造就出语言特性、程式库组件、程式技术之间令人眼花的大旋涡，既混沌又壮丽。就像真实程式一样。

我把 GotW 发音为"Gotcha"（意思是「这下可抓到你了」），或许很适当。当我把书中测验的（我的）答案拿来和 Sutter 的答案比较，我掉进他（和 C++）铺设的陷阱中— 虽然我实在不想承认这点。我几乎可以看见 Herb 微笑并温柔地对我所犯的每一个错误说"Gotcha!"。某些人可能会认为，这只不过证明我并不很了解 C++。也可能有人会因而主张 C++ 实在是太过复杂，精通它实在不容易。我相信这显示，当你选择 C++ 做为工具，你必须小心地思考你正在做些什么。C++ 是一个威力强大的语言，用来协助解决吃力的问题，其重要性使你必须尽可能面对语言本身、程式库、程式惯用手法来磨练你的知识。本书题目幅员广泛，可以协助你完成心愿。其独一无二的测验式教学也带来很大的帮助。

C++ 讨论群（newsgroups）上的老资格读者都知道，要成为"a Guru of the Week"（每周之星）是件多么不容易的事。老资格的参赛者可能体会更深。当然，网际网路上一星期只能产生一位 guru（大师级人物），不过，藉由本书之助，你可以合理地期望每次写程式都产出 guru 等级的码。

Scott Meyers
June 1999

3. 译序（jjhou）

C++ 是一个难学易用的语言!

C++ 的难学，不仅在其广博的语法，以及语法背後的语意，以及语意背後的深层思维，以及深层思维背後的物件模型;C++ 的难学,还在於它提供了四种不同(但相辅相成)的程式设计思维模式:procedural-based, object-based, object-oriented,generic paradigm。

世上没有白吃的午餐。又要有效率，又要有弹性，又要前瞻望远，又要回溯相容，又要能治大国，又要能烹小鲜，学习起来当然就不可能太简单。

在如此庞大复杂的机制下，万千使用者前仆後续的动力是：一旦学成，妙用无穷。C++ 相关书籍之多，车载斗量；如天上繁星，如过江之鲫。广博如四库全书者有之（The C++ Programming Language、C++ Primer），深奥如重山复水者有之（The Annotated C++ Reference Manual, Inside the C++ Object Model），细说历史者有之（The Design and Evolution of C++ , Ruminations on C++），独沽一味者有之（Polymorphism in C++, Genericity in C++），独树一帜者有之（DesignPatterns ,Large Scale C++ Software Design, C++ FAQs），程式库大全有之（The C++ Standard Library），另辟蹊径者有之（Generic Programming and the STL），工程经验之累积亦有之（Effective C++, More Effective C++, Exceptional C++）。

这其中,「工程经验之累积」对已具 C++ 相当基础的程式员而言，有著致命的吸引力与立竿见影的帮助。Scott Meyers 的 Effective C++ 和 More Effective C++ 是此类佼佼，Herb Sutter 的 Exceptional C++ 则是後起之秀。

这类书籍的一个共通特色是轻薄短小，并且高密度地纳入作者浸淫於 C++/OOP 领域多年而广泛的经验。它们不但开展读者的视野，也为读者提供各种 C++/OOP 常见问题或易犯错误的解决模型。某些小范围主题诸如「在 base

classes 中使用 virtual destructor」、「令 operator= 传回*this 的 reference」，可能在百科型 C++ 语言书籍中亦曾概略提过，但此类书籍以深度探索的方式，让我们了解问题背後的成因、最佳的解法、以及其他可能的牵扯。至於大范围主题，例如 smart pointers, reference counting, proxy classes，double dispatching, 基本上已属 design patterns 的层级！

这些都是经验的累积和心血的结晶。

我很高兴将以下三本极佳书籍，规划为一个系列，以精装的形式呈现给您：

1. Effective C++ 2/e, by Scott Meyers, AW 1998
2. More Effective C++, by Scott Meyers, AW 1996
3. Exceptional C++, by Herb Sutter, AW 1999

不论外装或内容，中文版比其英文版兄弟毫不逊色。本书不但保留原文本索引（见书後索引首页之说明），并加上精装、书签条、译注、书籍交叉参考[注 1]、完整范例码[注 2]、读者服务[注 3]。

这套书对於您的程式设计生涯，可带来重大帮助。制作这套书籍使我感觉非常快乐。我祈盼（并相信）您在阅读此书时拥有同样的心情。

侯捷 2000/11/15 于新竹.台湾
jjhou@ccca.nctu.edu.tw
http://www.jjhou.com
http://jjhou.readme.com.tw

注 1：Effective C++ 2/e 和 More Effective C++ 之中译，事实上是以 Scott Meyers 的另一个产品 Effective C++ CD 为本，不仅资料更新，同时亦将 CD 版中两书之交叉参考保留下来。这可为读者带来旁徵博引时的莫大帮助。

注 2：书中程式多为片段。我将陆续完成完整的范例程式，并在 Visual C++,C++Builder,GNU C++ 上测试。请至侯捷网站（http://www.jjhou.com）下载。

注 3：欢迎读者对本书范围所及的主题提出讨论，并感谢读者对本书的任何误失提出指正。来信请寄侯捷电子信箱（jjhou@ccca.nctu.edu.tw）。

4. 泛型程式设计与 C++ 标准程式库（Generic Programming and the C++ Standard Library）

一开始,让我们考虑泛型程式设计领域里头挑选出来的几个题目。这些难题的焦点放在如何有效使用 templates, iterators, algorithms，以及如何使用并扩充标准程式库的设施。然後，这些想法会漂亮地导引出下一个章节，分析撰写 exception-safe templates 时所谓的异常安全性（exception safety）。

条款 1：GotW#18 Iterators

困难度：7

每一位手上正在使用标准程式库的程式员，都必须知道以下这些常见（或不是那么常见）的 iterator 错误运用。你可以找出几个错误？

以下程式至少有四个与 iterator 相关的问题。你可以找出几个?

```
int main()
{
  vector<Date> e;
  copy ( istream_iterator<Date>(cin),
         istream_iterator<Date>(),
         back_inserter( e ) );
  vector<Date>::iterator first =
               find( e.begin(), e.end(), "01/01/95" );
  vector<Date>::iterator last =
               find( e.begin(), e.end(), "12/31/95" );
  *last = "12/30/95";
  copy( first,
        last,
        ostream_iterator<Date>( cout, "\n" ) );
  e.insert( --e.end(), TodaysDate() );
  copy( first,
```

```
        last,
        ostream_iterator( cout, "\n" ) );
}
```

解答

```
int main()
{
  vector<Date> e;
  copy ( istream_iterator<Date>(cin),
         istream_iterator<Date>(),
         back_inserter( e ) );
```

目前为止一切正确。Date class 的作者提供了一个萃取函式（extractor function），型式为 operator>>(istream&, Date&)，以便 istream_iterator<Date> 得以用来从 cin stream 读取 Dates 资料。上述的 copy 演算法会把 Dates 装填到 vector 内。

```
vector<Date>::iterator first =
                find( e.begin(), e.end(), "01/01/95" );
vector<Date>::iterator last =
                find( e.begin(), e.end(), "12/31/95" );
*last = "12/30/95";
```

错误：上一行可能是不合法的，因为 last 可能是 e.end()，因而不是一个可提领(dereferenceable)的 iterator。

如果找不到目标，find() 演算法会将其第二引数（也就是用以指示范围尾端者）传回。本例之中如果"12/31/95"不在 e 之内，那么 last 便等於 e.end()，也就是指向 container 最後一个元素的下一位置，那不是个有效的 iterator。

```
copy( first,last,ostream_iterator<Date>( cout, "\n" ) );
```

错误：这可能是不合法的，因为[first, last] 可能不是一个有效范围；因为 first 有可能在 last 之後。

举个例子，如果"01/01/95" 不在 e 之内而"12/31/95" 在 e 之内，那么 iterator last 将指向符合"12/31/95"的那个 Date object，而其位置将於 iterator first 之前；此时的 first 指向 e 的最後元素的下一位置。然而，copy 演算法要求 first 必须指向 last 之前，也就是说[first, last] 必须是一个有效范围（valid range）。除非你用的是一个有检验能力的标准程式库，可以侦测出像这样的问题，否则很可能在 copy() 演算法运算期间或运算之後，出现一个难以诊断的 core dump（译注：因程式错误而形成的一个错误状态档）。

```
e.insert( --e.end(), TodaysDate() );
```

错误之一：--e.end() 可能是不合法的。

理由很简单，但有一点隐晦：一般的 C++ 标准程式库实际上都是以一个 Date* 来实作出 vector<Date>::iterator，而 C++ 语言并不允许你修改内建型别的暂时物件。例如以下程式码并不合法：

```
Date* f(); // 函式传回一个 Date*
p = --f(); // 错误。但如果写为"f() - 1" 就可以。
```

幸运的是，我们知道 vector<Date>::iterator 是个随机存取（random access）的 iterator，所以修改成这样并不会损失效率：

```
e.insert( e.end() - 1, TodaysDate() );
```

错误之二：如果 e 是空的，任何人企图取得「e.end() 之前一个位置的 iterator」，不论是写成--e.end() 或是写成 e.end()-1，都不是有效的 iterator。

```
    copy( first,
          last,
          ostream_iterator( cout, "\n" ) );
}
```

错误：first 和 last 可能不是有效的 iterators。

vector 的成长系呈块状（chunks）成长，所以它不需要在你每次安插新元素时重新配置缓冲区。然而有时候 vector 呈满载状态，这时候再加入新元素进去，便会触发记忆体重新配置。

本例经过 e.insert() 之後，vector 可能成长也可能没有成长，意味其记忆体可能移动也可能没有移动。由於这种不确定性，我们必须视现有的任何 iterators 都不再有效。本例之中如果记忆体真的移动了，那么 copy() 会再次产生出难以诊断的 core dump。

设计准则：绝对不要提领（dereference）一个无效的 iterator。

摘要：使用 iterators 时，务必清楚以下四点：

1. 有效的数值：这个 iterator 可以提领吗？如果你写*e.end()，绝对是个错误。
2. 有效的寿命：这个 iterator 被使用时还有效吗？或是它已经因为某些操作而变得无效了。
3. 有效的范围：一对 iterators 是否组成一个有效范围？是否 first 真的在 last 之前（或相等）？是否两者指向同一个 container?
4. 不合法的操作行为：程式码是否企图修改内建型别的暂时物件，像--e.end() 这样？（幸运的是编译器通常能够捕捉这种错误。至於「指向 class 型别」（而非内建型别）的 iterator，程式库作者往往会允许这种事情发生，为

的是语法层面上的方便性）

条款 2～3：GotW#29 不区分大小写的 string （Case-Insensitive Strings）

难度：7/10

你期望一个不分大小写的字符串类型吗？你的使命是，应该选个现成的并接受它，还是自己写一个。

问题

写一个不分大小写的字符串类型，它其它方面都与标准库中的“string”类相同，只是在大小写区分上和（非标的，但被广泛使用的）C 函数 stricmp()：

```
ci_string s( "AbCdE" );
// case insensitive
assert( s == "abcde" );
assert( s == "ABCDE" );
// still case-preserving, of course
assert( strcmp( s.c_str(), "AbCdE" ) == 0 );
assert( strcmp( s.c_str(), "abcde" ) != 0 );
```

解决方案

写一个不分大小写的字符串类型，它其它方面都与标准库中的“string”类相同，只是在大小写区分上和（非标的，但被广泛使用的）C 函数 stricmp()：

“怎么实现一个不分大小写的字符串类型”这个问题是如此常见，以致于它需要一份专有的 FAQ－－所以才在 GotW 中讨论它。

注意 1：stricmp()这个不区分大小写的字符串比较函数不是 C 标准的一部分，但它为很多 C 编译器扩展提供。

注意 2：“不区分大小写”的实际含义完全取决于你的程序和国家语言。例如，很多语言根本就没有大小写；但即使如此，你仍然需要决策重读和非重读字符是否等价，诸如此类。

下面是我们期望达到的目标：本 GotW 指导了如何为标准 string 类实现“不区分大小写”，无论你处在什么语境下。

```
ci_string s( "AbCdE" );
// case insensitive
assert( s == "abcde" );
assert( s == "ABCDE" );
// still case-preserving, of course
assert( strcmp( s.c_str(), "AbCdE" ) == 0 );
assert( strcmp( s.c_str(), "abcde" ) != 0 );
```

关键点是领会“string 类”在标准 C++中到底是什么。如果你看一下 string 的头文件，你将看到如下的东西：

```
typedef basic_string<char> string;
```

所以，string 并不是一个真正的类，它是一个模板的（特化的）typedef。再向下，basic_string<>模板申明如下，这是其全貌：

```
template<class charT,
         class traits = char_traits<charT>,
         class Allocator = allocator<charT> >
    class basic_string;
```

所以，“string”实际上是“basic_string<char, char_traits<char>, allocator<char> >”。我们不必操心分配器（allocator）部分，关键点是 char_traits 部分，它决定了字符的相互作用和比较运算（!运算）。

basic_string 提供了常用的比较函数以比较两个 string 对象是否相等，或一个小于另一个，等等。这些 string 类的比较函数是建立在 char_traits 模板提供的字符比较函数基础上的。具体一点，char_traits 模板提供了如下的字符比较函数：eq()（相等）、ne()（不等）、lt()（小于）、compare()（比较字符序列）、find()（搜索字符序列）。

如果你希望在（string 的）这些操作上有不同的行为，我们所要做的只是提供一个不同的 char_traits 模板。这是最容易的方法：

```
struct ci_char_traits : public char_traits<char>
                // 继承为了得到我们不必过载的函数
{
  static bool eq( char c1, char c2 )
    { return toupper(c1) == toupper(c2); }
  static bool ne( char c1, char c2 )
    { return toupper(c1) != toupper(c2); }
  static bool lt( char c1, char c2 )
    { return toupper(c1) <  toupper(c2); }
  static int compare( const char* s1,
                      const char* s2,
                      size_t n ) {
    return memicmp( s1, s2, n );
```

```
                // 如果你的编译器提供了它，
                //  不然你就得自己实现一个。
        }
        static const char*
        find( const char* s, int n, char a ) {
          while( n-- > 0 && toupper(*s) != toupper(a) ) {
              ++s;
          }
          return s;
        }
      };
```

最后将它们合在一起：

```
  typedef basic_string<char, ci_char_traits> ci_string;
```

我们重定义了一个“ci_string”，它的操作非常象标准的“string”，只是它用 ci_char_traits 代替了 char_traits<char>以使用特别的比较规则。我们只不过将 ci_char_traits 的规则实现为“不区分大小写”，就使得 ci_string 大动手脚就表现为“不区分大小写”了——也就是说，我们根本没有碰 basic_string 就有了一个“不区分大小写”的 string!

这次的 GotW 揭示了 basic_string 模板的工作原理以及实现使用上的灵活性。如果你期望不用上面的 memicmp()和 toupper()实现的这些比较函数，只需要用你自己的代码替换这 5 个函数，怎么满足你自己的程序的需求，就怎么实现它们。

习题

1.这样从 char_traits<char>继承出 ci_char_traits 安全吗？为什么安全或为什么不安全？

2. 为什么下面的代码编译不通过？

（WQ 注，由于 C++的改进，此代码已经可以编译通过，并正常运行!）

```
    ci_string s = "abc";
    cout << s << endl;
```

提示 1：参见 GotW #19。

提示 2：摘自 21.3.7.9 [lib.string.io]，basic_string 的 operator<<操作申明如下（是个偏特化）：

```
    template<class charT, class traits, class Allocator>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os,
               const basic_string<charT,traits,Allocator>& str);
```

（WQ 注，C++标准库中，现在已将“basic_ostream<charT, traits>& os”改为“ostream&”，所以没问题了。）

ANSWER: Notice first that cout is actually a basic_ostream<char, char_traits<char> >. Then we see the problem: operator<< for basic_string is templated and all, but it's only specified for insertion into a basic_ostream with the same 'char type' and 'traits type' as the string. That is, the standard operator<< will let you output a ci_string to a basic_ostream<char, ci_char_traits>, which isn't what cout is even though ci_char_traits inherits from char_traits<char> in the above solution.

（由于已错误，不译。）

有两个解决办法：定义 ci_strings 自己的流入/流出函数，或使用“.c_str()”：

```
        cout << s.c_str() << endl;
```

3. 当在标准 string 对象和 ci_string 对象间使用其它操作（如+、+=、=）时，发生什么？例如：

```
        string    a = "aaa";
        ci_string b = "bbb";
        string    c = a + b;
```

答案：还是，定义 ci_string 自己的这些操作，或使用“.c_str()”：

```
    string    c = a + b.c_str();
```

条款 4～5：GotW#16 具有最大可复用性的通用 Containers（Maximally Reusable Generic Containers）

难度：8 / 10

（你能让这个简单的 container class 具有多大的可适应性（flexibility）？预告：在本条款中，你将要学到的关于成员模板方面的知识绝不只是一点点而已。）

[问题]

为下面的定长（fixed-length）vector class 实现拷贝构造（copy construction）操作和拷贝赋值（copy assignment）操作，以提供最大的可用性（usability）。提示：请考虑用户代码（client code）可能会用它做那些事情。

```
  template<typename T, size_t size>
  class fixed_vector {
  public:
```

```
    typedef T*       iterator;
    typedef const T* const_iterator;
    iterator       begin()       { return v_; }
    iterator       end()         { return v_+size; }
    const_iterator begin() const { return v_; }
    const_iterator end()   const { return v_+size; }
  private:
    T v_[size];
  };
```

[注意]

（1）不要修改代码的其它部分。给出这个 container 的目的并不是要将其纳入 STL——它至少有一个严重的问题；给出它的目的在于，我们能在一个简化了的情形下说明一些重要的问题。

（2）这里的代码是根据最初由 Kevlin Henney 给出的例子改编而来的。Jon Jagger 曾在 British C++ user magazine Overload 栏目的第 12 和 20 期上面分析过那个最初的例子。（英国的读者注意了：在本期 GotW 中所给出的解答与 British C++ user magazine Overload 第 20 期中的解答远不相同。事实上，在那一期中所提出的效率优化处理方案在我将给出的 GotW 解答中并不凑效。）

[解答]

[注意：这里的 GotW 原始解答存在一些 bug，但已经在 Exceptional C++和勘误表中解决了]

在本期 GotW 中，我们换一种方法来进行：由我来给出用于解答的代码，而由你来对代码进行解说。

提问：下面的解决方案具体是怎样运作的？为什么会这样运作？请对每一个构造函数（constructor）和运算符（operator）都作说明。

```
  template<typename T, size_t size>
  class fixed_vector {
  public:
    typedef T*       iterator;
    typedef const T* const_iterator;
    fixed_vector() { }
    template<typename O, size_t osize>
    fixed_vector( const fixed_vector<O,osize>& other ) {
      copy( other.begin(),
            other.begin()+min(size,osize),
            begin() );
    }
    template<typename O, size_t osize>
    fixed_vector<T,size>&
    operator=( const fixed_vector<O,osize>& other ) {
      copy( other.begin(),
            other.begin()+min(size,osize),
            begin() );
      return *this;
    }
    iterator       begin()       { return v_; }
    iterator       end()         { return v_+size; }
    const_iterator begin() const { return v_; }
    const_iterator end()   const { return v_+size; }
  private:
    T v_[size];
  };
```

现在，我们就来分析这段代码，看看它到底有没有圆满的回答本期的 GotW 问题。

[拷贝构造（Copy Contruction）操作和拷贝赋值（Copy Assignment）操作]

首先应该注意的是，本期条款的问题本身就有使用遮眼法（a red herring）的嫌疑——原始问题中给出的代码本身已经具有工作良好的拷贝构造函数（copy constructor）和拷贝赋值运算符（copy assignment operator）。而我们的解决方案意欲增加一个模板化的构造函数（templated constructor）和一个模板化的赋值运算符（templated assignment operator），以使得构造操作和赋值操作更具可适应性。

我要祝贺 Valentin Bonnard 以及其他一些人，是他们很快就指出，预想中的拷贝构造函数（copy constructor）其实压根儿就不是一个拷贝构造函数！事实上，连预想中的那个拷贝赋值运算符（copy assignment operator）其实也压根儿就不是一个拷贝赋值运算符！

其原因是：真正的拷贝构造函数或者拷贝赋值运算符只对完全相同类型的对象施以构造或赋值操作，并且，其如果是一个模板类的话，模板的参数也都必须完全相同。例如：

```
    struct X {
      template<typename T>
      X( const T& );    // 这不是拷贝构造函数，因为 T 不会是 X
```

```
  template<typename T>
  operator=( const T& );
                    //这不是拷贝赋值运算符，因为 T 不会是 X
};
```

“但是，”你说，“这两个被模板化了的成员函数的确具有拷贝构造（Copy Contruction）操作和拷贝赋值（Copy Assignment）操作的准确形式呀！”这个嘛……告诉你吧：其实不然——根本不是这回事，因为在那两种情况下，T 都不一定是 X。下面是摘自 CD2[注：同样的叙述也出现在 1998 年的官方标准中；“CD2”即“Committee Draft 2（委员会第 2 号草案）”]的叙述：

[12.8/2 note 4]

Because a template constructor is never a copy constructor, the presence of such a template does not suppress the implicit declaration of a copy constructor.

由于模板构造函数终究不是拷贝构造函数，因此这种模板的出现并不会隐藏原来隐含的拷贝构造函数之声明。

在[12.8/9 note 7]中也有关于拷贝赋值操作的类似叙述。如此一来，我们在解答中给出的代码实际上与原来问题中的原始代码有着相同的拷贝构造函数和拷贝赋值运算符——因为编译器始终生成它们隐含的版本。我们所做的改动只是增强了构造操作和赋值操作的可适应性，而不是替换掉了旧有的版本。举个例子来说：

```
fixed_vector<char,4> v;
fixed_vector<int,4>  w;
fixed_vector<int,4>  w2(w);
        // 调用隐含的拷贝构造函数
fixed_vector<int,4>  w3(v);
        // 调用模板化了的转换构造函数
w = w2; // 调用隐含的赋值运算符
w = v;  // 调用模板化了的赋值运算符
```

由此可以看出，本条款的问题所寻求的真正答案其实是提供了具有可适应性的“从其它 fixed_vectors 进行构造和拷贝的操作”，而不是具有可适应性的“拷贝构造操作和拷贝赋值操作”——它们早就存在了。

[构造操作和赋值操作的可用性]

我们增加的两个操作具有如下两个主要用途：

1. 支持可变的类型（包括继承在内）

尽管 fixed_vector 原则上应该保持在相同类型的 container 之间进行拷贝和赋值操作，但有时候从另一个包含不同类型的对象之 fixed_vector 进行构造和赋值操作，也是不无意义的。只要源对象可以被赋值给目的对象，就应该允许这种不同类型对象之间的赋值。例如，用户代码可能会这样使用 fixed_vector：

```
fixed_vector<char,4> v;
fixed_vector<int,4>  w(v);  // 拷贝
w = v;                      // 赋值
class B { /*...*/ };
class D : public B { /*...*/ };
fixed_vector<D,4> x;
fixed_vector<B,4> y(x);     // 拷贝
y = x;                      // 赋值
```

2. 支持可变的大小

与第 1 点类似，用户代码有时也可能希望从具有不同大小的 fixed_vector 进行构造或赋值。支持这个操作同样也是有意义的。例如：

```
fixed_vector<char,6> v;
fixed_vector<int,4>  w(v);  // 拷贝 4 个对象
w = v;                      // 对 4 个对象进行赋值
class B { /*...*/ };
class D : public B { /*...*/ };
fixed_vector<D,16> x;
fixed_vector<B,42> y(x);    // 拷贝 16 个对象
y = x;                      // 对 16 个对象进行赋值
```

[另一种解决方案：标准库风格的解答]

我很喜爱以上函数的形式以及其甚佳的可用性，但它们还是有些做不到的事情。接下来，我们考察一种具有标准库风格的解答：

1. 拷贝（Copying）

```
template<Iter>
fixed_vector( Iter first, Iter last ) {
  copy( first,
        first+min(size,last-first),
        begin() );
}
```

于是，当要进行拷贝操作时，我们不用：

```
fixed_vector<char,6> v;
```

```
    fixed_vector<int,4>  w(v);  // 拷贝 4 个对象
```
而要用：
```
    fixed_vector<char,6> v;
    fixed_vector<int,4>  w(v.begin(), v.end());
                                // 拷贝 4 个对象
```
对于一个构造操作而言，哪一种方案更好呢：是先前的预想方案，还是这个标准库风格的方案？前一个相对更容易使用一些，而后一个的可适应性更好（比如它允许用户选择操作范围并可以其它种类的 container 进行拷贝）——你可以选择任一种或者干脆两者兼而提供之。

2. 赋值（Assignment）

这里应注意的是，由于 operator=()只能接收一个参数，因此我们无法让赋值操作把 iterator 的范围作为另一个参数。一个可行的方法是，提供一个具名函数（named function）：
```
    template<Iter>
    fixed_vector<T,size>&
    assign( Iter first, Iter last ) {
      copy( first,
            first+min(size,last-first),
            begin() );
      return *this;
    }
```
于是，当要进行赋值操作时，我们不用：
```
    w = v;                      // 对 4 个对象进行赋值
```
而要用：
```
    w.assign(v.begin(), v.end());
                                // 对 4 个对象进行赋值
```
从技术上而言，assign()其实并不是必需的，没有它我们也可以达到相同程度的可适应性，但这样一来，进行赋值操作时就不太有效率，显得比较不爽：
```
    w = fixed_vector<int,4>(v.begin(), v.end());
                                // 对 4 个对象进行赋值
```
对于一个赋值操作而言，哪一种方案更好呢：是先前的预想方案，还是这个标准库风格的方案？这一回我们就不能再像第 1 点中那样用其可适应性来衡量孰好孰坏了，因为用户可以很容易的编写拷贝操作的代码（这样甚至还更具可适应性）。用户将不会用：
```
    w.assign(v.begin(), v.end());
```
而会直接用：
```
    copy( v.begin(), v.end(), w.begin() );
```
在这种情况下，就没有必要编写 assign()了。因此使用先前预想的方案或许更好一些，这样可以让用户代码在需要对某个范围内的对象进行赋值操作的时候直接使用 copy()。

[为什么给出了缺省构造函数（default constructor）？]

最后的问题：既然第一种预想方案中的空的缺省构造函数之功能与编译器自己生成的缺省构造函数之功能相同，那为什么还要特意在解答代码中给出它来呢？这是因为，一旦你定义了一个任意形式的构造函数，编译器就不会为你生成其缺省版本了。显然，像上述那样的用户代码就需要这样做。

[小结：成员函数模板（Member Function Templates）到底怎么样？]

希望本期 GotW 条款已经使你确信，成员函数模板非常易用。另外，我还希望本期条款能够使你明白为什么成员函数模板会被广泛用于标准库中。如果你还对此用法不太熟悉，千万不要伤心欲绝——并不是所有现存的编译器都支持成员模板特性，只不过这是 C++标准的规定，所有的编译器很快都将支持它。（在撰写本文时，Microsoft Visual C++ 5.0 已经可以编译通过使用此特性的代码了，但在某些用户代码例程中，其编译器还是不能对 osize 参数进行推理分析。）

在你创建自己的 classes 时使用成员模板，不仅可以取悦用户，而且用户还会越来越多，并争先恐后的使用那些极易复用的代码。

条款 6：GotW#2 临时对象（Temporary Objects）

难度：5 / 10

（把你的心血之作（包括你的程序之性能在内）当成垃圾抛出窗外的罪人，往往是一些意想不到的临时对象。）

[问题]

试想你正在阅读另一个程序员写好的函数代码（如下），而这个函数中却在至少三个地方用到了不必要的临时对象。那么，你能发现其中的几个呢？程序员又该如何修改代码呢？
```
    string FindAddr( list<Employee> l, string name )
    {
      for( list<Employee>::iterator i = l.begin();
           i != l.end();
           i++ )
```

```
    {
      if( *i == name )
      {
        return (*i).addr;
      }
    }
    return "";
  }
```

[解答]

信不信由你，这短短的几行代码中就起码有三个地方明显的使用了不必要的临时对象，其中有两个比较微妙，第三个则是一计遮眼法（red herring）。

```
*       string FindAddr( list<Employee> l, string name )
---- 第 1 处 ----     ---- 第 2 处 ----
```

1 和 2：两个参数都应该使用常量引用（const reference）。使用传值（pass-by-value）方式将会导致函数对 list 和 string 进行拷贝，其性能代价是高昂的。

[规则]：请使用 const&而不是传值拷贝。

```
*         for( list<Employee>::iterator i = l.begin();
                i != l.end();
        i++ )
   - 第 3 处 -
```

3：这一处真是更为微妙。先增（preincrement）操作比后增（postincrement）操作效率更高，这是因为在进行后增（postincrement）操作时，对象不但必须自己递增，而且还要返回一个包含递增前之值的临时对象。要知道，就连 int 这样的内建类型也是如此！

[学习指导]：请使用先增（preincrement）操作，避免使用后增（postincrement）操作。

```
*             if( *i == name )
                                        -- 第 4 处 --
```

4：这里没有体现 Employee 类，但如果想让它行得通，则要么来一个转换成 string 的操作，要么通过一个转换构造函数（constructor）来得到一个 string。然而两种方法都会产生临时对象，从而导致对 string 或者 Employee 的 operator=之调用。()

[学习指导]：时刻注意因为参数转换操作而产生的隐藏的临时对象。一个避免它的好办法就是尽可能显式(explicit)的使用构造函数（constructor)。

```
*       return "";
         -- 第 5 处 --
```

5：这里产生了一个临时的（空的）string 对象。更好的做法是，声明一个局部 string 对象来储存返回值，然后用单独一个 return 语句返回这个 string。这使得编译器可以在某些情况下（比如，形如“string a = FindAddr(l, “Harold”);”的代码）启用“返回值优化”处理来省略掉局部对象。

[规则]：请遵循所谓的“单入口/单出口”(single-entry/single-exit)规则。绝不要在一个函数里面写有多个 return 语句。

[作者记：当进行了进一步的性能测试之后，我不再认同上面的那条建议。我已经在《Exceptional C++》中修改了这一点。]

```
*      string FindAddr( list<Employee> l, string name )
--          第 * 处 --
```

* ：这可是一计遮眼法（red herring）。看上去，好像你可以很简单的通过把返回类型声明为 string&而不是 string，来避免在所有可能的返回问题中产生临时对象。对吗？错了！如果你的程序只是在代码试图使用引用（reference）的时候就崩溃（因为那个所指向的局部对象早已不存在），那就算你够走运的了！如果你不走运的话，你的代码将看上去似乎能够正常工作，却时不时的冷不防失败几次，从而使你不得不在调试程序的过程中渡过一个又一个漫漫长夜。

[规则]：绝对绝对（!）不要返回对局部对象的引用（reference）。

[作者记：有一些贴子正确的指出，你可以声明一个遇到错误时才返回的静态对象，从而实现在不改变函数语义的情况下返回一个引用（reference）。同时这也意味着，在返回引用（reference）的时候，你必须注意对象的生存周期。]

其实还有很多可以优化的地方，诸如“避免对 end()进行多余的调用”等等。程序员可以（也应该）使用一个 const_iterator。抛开这些不谈，我们仍可以得到如下的正确代码：

```
string FindAddr( const list<Employee>& l, const string& name )
{
    string addr;
    for( list<Employee>::const_iterator i = l.begin();
        i != l.end();
        ++i )
  {
```

```
                if( (*i).name == name )
                 {
               addr = (*i).addr;
  break;
                 }
          }
          return addr;
        }
```

[kingofark 注：
red herring: Something that draws attention away from the central issue.
a red herring: 遮眼法; 转移注意力的东西]

条款 7：GotW#3 使用标准库（Using the Standard Library）

难度：3 / 10

（使用标准库的算法要比使用自己徒手编写的算法实现好得多。这里我们将重新使用 GotW 条款 02 中的例子，借以说明，通过简单的复用（reuse）那些标准库里面已有的东西，将会避免多少麻烦的问题。）

[问题]

一旦程序员使用标准库算法而不是编写自己的算法版本时，首先就可以避免 GotW 条款 02 中的多少个缺陷？（注意：和上次一样，不能改变函数的语义，即使这样做可以改良函数本身。）

[版本翻新]：

原始的错误版本：

```
string FindAddr( list<Employee> l, string name )
{
  for( list<Employee>::iterator i = l.begin();
       i != l.end();
       i++ )
  {
    if( *i == name )
    {
      return (*i).addr;
    }
  }
  return "";
}
```

修正的版本（仍然存在对 l.end()多余的调用）：

```
string FindAddr( const list<Employee>& l,
                 const string& name )
{
  string addr;
  for( list<Employee>::const_iterator i = l.begin();
       i != l.end();
       ++i )
  {
    if( (*i).name == name )
    {
      addr = (*i).addr;
      break;
    }
  }
  return addr;
}
```

[解答]

不需要什么别的改变，只要简单的使用 find()就可以避免产生两个临时对象和原始版本中几乎所有 l.end()带来的低效：

```
string FindAddr( list<Employee> l, string name )
{
  list<Employee>::iterator i =
    find( l.begin(), l.end(), name );
  if( *i != l.end() )
  {
```

```
    return (*i).addr;
  }
  return "";
}
```

再进行一些其它的修正，得到：

```
string FindAddr( const list<Employee>& l,
                 const string& name )
{
  string addr;
  list<Employee>::const_iterator i =
    find( l.begin(), l.end(), name );
  if( i != l.end() )
  {
    addr = (*i).addr;
  }
  return addr;
}
```

[学习指导]：请复用标准库的算法，而不要编写自己的算法版本。这样做更快、更容易、更安全！

5. Exception-Safety 的主题与相关技术（Exception-Safety Issues and Techniques）

条款 8～17：GotW#8 挑战篇——异常处理的安全性（Challenge Edition - Exception Safety）

难度：9 / 10

（异常处理机制是解决某些问题的上佳办法，但同时它也引入了许多隐藏的控制流程；有时候，要正确无误的使用它并不容易。不妨试试自己实现一个简单的 container（这是一种可以对其进行 push 和 pop 操作的栈），看看它在异常-安全的（exception-safe）和异常-中立的（exception-neutral）情况下，到底会发生哪些事情。）

[问题]

1. 实现如下异常-中立的（exception-neutral）container。要求：Stack 对象的状态必须保持其一致性（consistent）；即使有内部操作抛出异常，Stack 对象也必须是可析构的（destructible）；T 的异常必须能够传递到其调用者那里。

```
template <class T>
    // T 必须有缺省的构造函数和拷贝构造函数
class Stack
{
public:
    Stack();
    ~Stack();
    Stack(const Stack&);
    Stack& operator=(const Stack&);
    unsigned Count();   // 返回 T 在栈里面的数目
    void     Push(const T&);
    T        Pop();     // 如果为空，则返回缺省构造出来的 T
private:
    T*       v_;        // 指向一个用于'vsize_' T 对象的
                        //  足够大的内存空间
    unsigned vsize_;    // 'v_' 区域的大小
    unsigned vused_;    // 'v_' 区域中实际使用的 T 的数目
};
```

附加题：

2. 根据当前的 C++标准，标准库中的 container 是异常-安全的（exception-safe）还是异常-中立的（exception-neutral）？

3. 应该让 container 成为异常-中立的（exception-neutral）吗？为什么？有什么折衷方案吗？

4. Container 应该使用异常规则吗？比如，我们到底应不应该作诸如“Stack::Stack()throw(bad_alloc);”的声明？

挑战极限的问题：

5. 由于在目前许多的编译器中使用 try 和 catch 会给你的程序带来一些额外的负荷，所以在我们这种低级的可复用（reusable）Container 中，最好避免使用它们。你能在不使用 try 和 catch 的情况下，按照要求实现 Stack 所有的成员函数吗？

在这里提供两个例子以供参考（注意这两个例子并不一定符合上述题目中的要求，仅供参考，以便于你下手解题）：

```
template<class T>
Stack<T>::Stack()
  : v_(0),
    vsize_(10),
    vused_(0)
{
    v_ = new T[vsize_]; // 初始的内存分配（创建对象）
}
template<class T>
T Stack<T>::Pop()
{
    T result; //如果为空，则返回缺省构造出来的 T
    if( vused_ > 0)
    {
        result = v_[--vused_];
    }
    return result;
}
```

[解答]

[作者记：这里的解决方案并不完全正确。本文经修正的增强版本，你可以在 C++Report 1997 年的 9 月号、11 月号和 12 月号上面找到；另外，其最终版本在我的《Exceptional C++》里面。]

重要的事项：我确实不敢保证下面的解决方案完全满足了我原题的要求。实际上我连能够正确编译这些代码的编译器都找不到！在这里，我讨论了所有我能想得到的那些交互作用；而本文的主要目的则是希望说明，在编写异常-安全的（exception-safe）代码时需要格外的小心。

另外，Tom Cargill 也有一篇非常棒的文章《Exception Handling:A False Sense of Security》(C++Report, vol.9 no.6, Nov-Dec 1994)。他通过这篇文章来说明，异常处理是个棘手的小花招，技巧性非常强，但也并不就是笼统的说不要使用异常处理，而是说人们不要过分的迷信异常处理。只要认识到这一点，并在使用时小心一点就可以了。

[作者再记：最后再说一点。为了简化解决方案，我决定不去讨论用来解决异常-安全（exception-safe）资源之归属问题的基类技术（base class technique）。我会邀请 Dave Abrahams（或者其他人）来继续讨论，阐述这个非常有效的技术。]

现在先回顾一下我们的问题。需要的接口如下：

```
template <class T>
    // T 必须有缺省的构造函数和拷贝构造函数
class Stack
{
public:
    Stack();
    ~Stack();
    Stack(const Stack&);
    Stack& operator=(const Stack&);
    unsigned Count();   //返回 T 在栈里面的数目
    void     Push(const T&);
    T        Pop();     //如果为空，则返回缺省构造出来的 T
private:
    T*       v_;        //指向一个用于'vsize_' T 对象的
                        //  足够大的内存空间
    unsigned vsize_;    // 'v_'区域的大小
    unsigned vused_;    // 'v_' 区域中实际使用的 T 的数目
};
```

现在我们来看看实现。我们对 T 有一个要求，就是 T 的析构函数（destructor）不能抛出异常。这是因为，如果允许 T 的析构函数（destructor）抛出异常，那我们就很难甚至是不可能在保证代码安全性的前提下进行实现了。

```
//----- DEFAULT CTOR --------------------------------------------
template<class T>
Stack<T>::Stack()
  : v_(new T[10]),  // 缺省的内存分配（创建对象）
    vsize_(10),
    vused_(0)       // 现在还没有被使用
{
    // 如果程序到达这里，说明构造过程没有问题，okay!
```

```
}
//----- 拷贝构造函数 ----------------------------------------------------
template<class T>
Stack<T>::Stack( const Stack<T>& other )
  : v_(0),      // 没分配内存，也没有被使用
    vsize_(other.vsize_),
    vused_(other.vused_)
{
    v_ = NewCopy( other.v_, other.vsize_, other.vsize_ );
    //如果程序到达这里，说明拷贝构造过程没有问题，okay!
}
//----- 拷贝赋值 --------------------------------------------
template<class T>
Stack<T>& Stack<T>::operator=( const Stack<T>& other )
{
    if( this != &other )
    {
        T* v_new = NewCopy( other.v_, other.vsize_, other.vsize_ );
        //如果程序到达这里，说明内存分配和拷贝过程都没有问题，okay!
        delete[] v_;
        // 这里不能抛出异常，因为 T 的析构函数不能抛出异常；
        // ::operator delete[] 被声明成 throw()
        v_ = v_new;
        vsize_ = other.vsize_;
        vused_ = other.vused_;
    }
    return *this;   // 很安全，没有拷贝问题
}
//----- 析构函数 --------------------------------------------------------
template<class T>
Stack<T>::~Stack()
{
    delete[] v_;    // 同上，这里也不能抛出异常
}
//----- 计数 ------------------------------------------------------------
template<class T>
unsigned Stack<T>::Count()
{
    return vused_;  // 这只是一个内建类型，不会有问题
}
//----- push 操作 -------------------------------------------------------
template<class T>
void Stack<T>::Push( const T& t )
{
    if( vused_ == vsize_ )  // 可以随着需要而增长
    {
        unsigned vsize_new = (vsize_+1)*2; // 增长因子
        T* v_new = NewCopy( v_, vsize_, vsize_new );
        //如果程序到达这里，说明内存分配和拷贝过程都没有问题，okay!
        delete[] v_;    //同上，这里也不能抛出异常
        v_ = v_new;
        vsize_ = vsize_new;
    }
    v_[vused_] = t; // 如果这里抛出异常，增加操作则不会执行，
    ++vused_;       //  状态也不会改变
}
//----- pop 操作 --------------------------------------------------------
template<class T>
T Stack<T>::Pop()
{
    T result;
    if( vused_ > 0)
```

```
    {
        result = v_[vused_-1];  //如果这里抛出异常，相减操作则不会执行，
        --vused_;               //  状态也不会改变
    }
    return result;
}
//
// 注意: 细心的读者 Wil Evers 第一个指出，
//  “正如在问题中定义的那样， Pop()强迫使用者编写非异常-安全的代码，
//  这首先就产生了一个负面效应（即从栈中间 pop 出一个元素）；
//  其次，这还可能导致遗漏某些异常（比如将返回值拷贝到代码调用者的目标
//  对象上）。”
//
// 同时这也表明，很难编写异常-安全的代码的一个原因就是因为
// 它不仅影响代码的实现部分，而且还会影响其接口！
// 某些接口（比如这里的这一个）不可能在完全保证异常-安全的情况下被实现。
//
// 解决这个问题的一个可行方法是把函数重新构造成
// "void Stack<T>::Pop( T& result)".
// 这样，我们就可以在栈的状态改变之前得知到结果的拷贝是否真的成功了。
// 举个例子如下，
// 这是一个更具有异常-安全性的 Pop()
//
template<class T>
void Stack<T>::Pop( T& result )
{
    if( vused_ > 0)
    {
        result = v_[vused_-1];  //如果这里抛出异常，
        --vused_;               //  相减操作则不会执行，
    }                           //  状态也不会改变
}
//
// 这里我们还可以让 Pop()返回 void，然后再提供一个 Front() 成员函数，
// 用来访问顶端的对象
//
//----- 辅助函数 ----------------------------------------------
// 当我们要把 T 从缓冲区拷贝到一个更大的缓冲区时，
//  这个辅助函数会帮助分配新的缓冲区，并把元素原样拷贝过来。
//  如果在这里发生了异常，辅助函数会释放占用得所有临时资源，
//  并把这个异常传递出去，保证不发生内存泄漏。
//
template<class T>
T* NewCopy( const T* src, unsigned srcsize, unsigned destsize )
{
    destsize = max( srcsize, destsize ); // 基本的参数检查
    T* dest = new T[destsize];
    // 如果程序到达这里，说明内存分配和构造函数都没有问题，okay!
    try
    {
        copy( src, src+srcsize, dest );
    }
    catch(...)
    {
        delete[] dest;
        throw;  // 重新抛出原来的异常
    }
    // 如果程序达到这里，说明拷贝操作也没有问题，okay!
    return dest;
}
```

对附加题的解答：

第 2 题：根据当前的 C++标准，标准库中的 container 是异常-安全的（exception-safe）还是异常-中立的

（exception-neutral）？

关于这个问题，目前还没有明确的说法。最近委员会也展开了一些相关的讨论，涉及到应该提供并保证弱异常安全性（即“container 总是可以进行析构操作”）还是应该提供并保证强异常安全性（即“所有的 container 操作都要从语义上具有‘要么执行要么撤销（commit-or-rollback）’的特性”）。正如 Dave Abrahams 在委员会中的一次讨论以及随后通过电子邮件进行的讨论中所表明的那样，如果实现了对弱异常安全性的保证，那么强异常安全性也就很容易得到保证了。我们在上面提到的几个操作正是这样的。

第 3 题：应该让 container 成为异常-中立的（exception-neutral）吗？为什么？有什么折衷方案吗？

有时候，为了保证某些 container 异常-中立性（exception-neutrality），其内的某些操作将会不可避免的付出一些空间代价。可见异常-中立性（exception-neutrality）本身并不错，但是当实现强异常安全性所要付出的空间或时间代价远远大于实现弱异常安全性的付出的时候，要实现异常-中立性（exception-neutrality）就太不现实了。有一个比较好的折衷方案，那就是用文档记录下 T 中不允许抛出异常的操作，然后通过遵守这些文档规则来保证其异常-中立性（exception-neutrality）。

第 4 题:Container 应该使用异常规则吗？比如,我们到底应不应该作诸如“Stack::Stack()throw(bad_alloc);”的声明？

答案是否定的。我们不能这样做，因为我们预先并不知道 T 中哪些操作会抛出异常，也不知道会抛出什么样的异常。

应该注意的是，有些 container 的某些操作（例如，Count()）只是简单的返回一个数值，所以我们可以断定它不会抛出异常。虽然我们原则上可以用 throw()来声明这类操作，但是最好不要这么做；原因有两个：第一，如果你这样做了，那么当你以后想修改实现细节使其可以抛出异常的时候，就会发现其存在着很大的限制；第二，无论异常是否被抛出，异常声明（exception specification）都会带来额外的性能开销。因此，对于那些频繁使用的操作，最好不要作异常声明（exception specification）以避免这种性能开销。

对挑战极限题的解答：

第 5 题：由于在目前许多的编译器中使用 try 和 catch 会给你的程序带来一些额外的负荷，所以在我们这种低级的可复用（reusable）Container 中，最好避免使用它们。你能在不使用 try 和 catch 的情况下，按照要求实现 Stack 所有的成员函数吗？

是的，这是可行的，因为我们仅仅只需要捕获“...”部分（见下面的代码）。一般，形如

```
try { TryCode(); } catch(...) { CatchCode(parms); throw; }
```

的代码都可以改写成这样：

```
struct Janitor {
    Janitor(Parms p) : pa(p) {}
    ~Janitor() { if uncaught_exception() CatchCode(pa); }
    Parms pa;
};
{
    Janitor j(parms); // j is destroyed both if TryCode()
                      // succeeds and if it throws
    TryCode();
}
```

我们只在 NewCopy 函数中使用了 try 和 catch。下面就是重写的 NewCopy 函数，用以体现上面说的改写技术：

```
template<class T>
T* NewCopy( const T* src, unsigned srcsize, unsigned destsize )
{
    destsize = max( srcsize, destsize ); // basic parm check
    struct Janitor {
        Janitor( T* p ) : pa(p) {}
        ~Janitor() { if( uncaught_exception() ) delete[] pa; }
        T* pa;
    };
    T* dest = new T[destsize];
    // if we got here, the allocation/ctors were okay
    Janitor j(dest);
    copy( src, src+srcsize, dest );
    // if we got here, the copy was okay... otherwise, j
    // was destroyed during stack unwinding and will handle
    // the cleanup of dest to avoid leaking memory
    return dest;
}
```

我已经说过，我曾与几个擅长靠经验来进行速度测试的人讨论过上述问题。结论是在没有异常发生的情况下，try 和 catch 往往要比其它方法快得多，而且今后还可能变得更快。但尽管如此，这种避免使用 try 和 catch 的技术还是非常重要的，一来是因为有时候就是需要写一些比较规整、比较容易维护的代码；二来是因为现有的一些编译器在处理 try 和 catch 的时候，无论在产生异常的情况下还是在不产生异常的情况下，都会生成效率极其低下的代码。

条款 18：GotW#20 代码的复杂性（Code Complexity）（一）

难度：9 / 10

（本条款提出了一个有趣味的挑战：在一个简单得只有三行代码的函数里可以有多少条执行路经？其答案几乎将肯定让你吃惊。）

[问题]

在没有任何其它附加信息的情况下，下列代码中可以有多少条执行路经？

```
String EvaluateSalaryAndReturnName( Employee e )
{
  if( e.Title() == "CEO" || e.Salary() > 100000 )
  {
    cout << e.First() << " " << e.Last()
         << " is overpaid" << endl;
  }
  return e.First() + " " + e.Last();
}
```

[解答]

假设：

a)忽略对函数参数求值时的不同顺序以及由析构函数（destructor）抛出的异常。[注 1]

下面的问题提给无所畏惧的勇者：

如果允许析构函数抛出异常，那么共会有多少条执行路经呢？

b) 调用的函数被认为具有原子性。事实上，例如”e.Title()”这个调用就可能由于好几个原因而抛出异常（比如，它自己本身可能抛出异常；它也可能由于「未能捕获由其调用的另一个函数所抛出的异常」而抛出异常；或者它可能采用 return by value（传值返回）方式从而造成临时对象得构造函数可能抛出异常）。这里我们假设对于函数而言，只关注执行 e.Title()操作的结果，即完成该操作后是否抛出了异常。

解答：23（仅仅在 4 行代码里!）

如果你找到了	给自己评等级
3	平均水平（Average）
4-14	能够认知异常（Exception-Aware）
15-23	精英资质（Guru Material）

这 23 条执行路径包括：

——3 条与异常无关的（non-exceptional）路径

——20 条暗藏的路径，都与异常有关

要理解那 3 条普通路径，诀窍就是要知道 C/C++的“短路求值规则（Short-Circuit Evaluation Rule)”：

1. 如果 e.Title()==”CEO”，那么就不需要对第二个条件求值了（比如，e.Salary()将不会被调用），但 cout 还是会被执行的。[注 2]

2. 如果 e.Title()!=”CEO”但 e.Salary()>100000，那么两个条件都会被求值，cout 也会被执行。

3. 如果 e.Title()!=”CEO”且 e.Salary()<=100000，那么 cout 将不会被执行。

下述都是由异常引出的执行路径：

```
String EvaluateSalaryAndReturnName( Employee e )
  ^*^                                      ^4^
```

4. 引数采用 pass by value（值传递）方式，这将唤起 Employee copy constructor。这个 copy 操作可能抛出异常。

*. 在将函数临时的返回值拷贝到函数调用者的区域时，String 的 copy constructor 可能抛出异常。然而在这里我们忽略这种可能性，因为其是在函数外部发生的(何况从目前的情形来看，现有的执行路径已经够我们忙的了!)

```
  if( e.Title() == "CEO" || e.Salary() > 100000 )
       ^5^     ^7^  ^6^ ^11^   ^8^    ^10^  ^9^
```

5. 成员函数 Title()本身就可能抛出异常；或者其采用 return by value 方式返回 class type 的对象，从而导致拷贝操作可能抛出异常。

6. 为了与有效的 operator==相匹配，字符串也许需要被转换成 class type（或许与 e.Title()的返回型别相同）的临时对象，而这个临时对象的构造过程可能抛出异常。

7. 如果 operator==是由程序员提供的函数，那么它可能抛出异常。

8. 与#5 类似，Salary()本身可能抛出异常，或者由于其返回临时对象而造成在临时对象的构造过程中抛出异常。

9. 与#6 类似，可能需要构造临时对象，而这个构造过程可能抛出异常。

10. 与#7 类似，这或许是由程序员提供的函数，那么它可能抛出异常。

11. 与#7 和#10 类似，这或许是由程序员提供的函数，那么它可能抛出异常。

```
  cout << e.First() << " " << e.Last()
          ^17^                ^18^
       << " is overpaid" << endl;
```

12-16. 如 C++标准草案所述，这里的五个对 operator<<的调用都可能抛出异常。

17-18. 与#5 类似。First()和/或 Last()可能抛出异常，或者由于其返回临时对象而造成在对象的构造过程中可能抛出异常。

```
return e.First()  +  " "   +   e.Last();
                ^19^  ^22^^21^ ^23^  ^20^
```

19-20. 与#5 类似。First()和/或 Last()可能抛出异常，或者由于其返回临时对象而造成在对象的构造过程中可能抛出异常。

21. 与#6 类似，可能需要构造临时对象，而这个构造过程可能抛出异常。

22-23. 与#7 类似，这或许是由程序员提供的函数，那么它可能抛出异常。

本期 GotW 条款的目的是演示「在一个允许异常机制的语言中，简单的代码里可以存在多少条暗藏的执行路径」。这种暗藏的复杂性会影响函数的可靠性和可测性吗？请在下一期 GotW 中寻找这个问题的答案。

[注 1]：决不允许一个异常从析构函数中渗透出来。如果允许这样做，代码将无法正常工作。请看我在 C++Report Nov/Dec 1997 中有关的更多讨论：Destructors That Throw and Why They're Evil。

[注 2]：如果对==、||和>予以正确恰当的重载（overload），那么在 if 语句中，||或许是一个函数调用。如果其是一个函数调用，那么“短路求值规则”会被抑住，这样 if 语句中的所有条件将总是被求值。

条款 19：GotW#21 代码的复杂性（Code Complexity）（二）

难度：7 / 10

（大挑战：修改 GotW#20 中那个只有三行代码的函数，使其成为强异常安全的（strongly exception-safe）。这个练习给我们上了关于异常安全性（exception-safety）的重要一课。）

[问题]

让我们来考虑 GotW#20 里的那个函数。这个函数是异常安全的(exception-safe)（再出现异常时仍能正常工作）还是异常中立的（exception-neutral)（能将所有异常都转给调用者）？

```
String EvaluateSalaryAndReturnName( Employee e )
{
  if( e.Title() == "CEO" || e.Salary() > 100000 )
  {
    cout << e.First() << " " << e.Last()
         << " is overpaid" << endl;
  }
  return e.First() + " " + e.Last();
}
```

请对你的回答做出解释。如果它是异常安全的，那它是支持 basic guarantee 还是支持 strong guarantee？[译注：basic guarantee，基本保证；strong guarantee，强力保证] 如果它不是异常安全的，那该如何对其进行修改以使其支持 basic guarantee 或 strong guarantee？

这里我们假设所有被调用的函数都是异常安全的（即可能抛出异常，但在抛出异常时没有任何副作用），并且假设所使用的任何对象（包括临时对象在内）也都是异常安全的（即当这些对象被销毁时，其占用的资源也能被清理）。

[背景知识：Basic Guarantee 和 Strong Guarantee]

关于 basic guarantee 和 strong guarantee 的详细论述，请看我在 C++ Report Sep/Nov/Dec 1997 中的文章。简单来说，basic guarantee 保证可销毁性（destructibility）且没有泄漏；而 strong guarantee 除此之外还保证完全的 commit-or-rollback（译注：即“要么执行，要么不执行”的原子规则）语义。

[解答]

让我们来考虑 GotW#20 里的那个函数。这个函数是异常安全的(exception-safe)（再出现异常时仍能正常工作）还是异常中立的（exception-neutral)（能将所有异常都转给调用者）？

```
String EvaluateSalaryAndReturnName( Employee e )
{
  if( e.Title() == "CEO" || e.Salary() > 100000 )
  {
    cout << e.First() << " " << e.Last()
         << " is overpaid" << endl;
  }
  return e.First() + " " + e.Last();
}
```

[关于假设的一点说明]

如题所述，我们假设所有被调用的函数——包括流函数（stream function）在内——都是异常安全的（即可能抛出异常，但在抛出异常时无副作用），并且假设所使用到的所有对象——包括临时对象在内——也都是异常安全的（即当这些对象被销毁时，其占用的资源也都能被清理）。

然而流（stream）却偏偏要对此使个拌儿——这缘于其可能产生的“un-rollbackable（不可回复）”副作用。例如，运算符<<可能会在输出（emitting）了 string 的一部分之后抛出一个异常，而此时已经被输出的那部分是无

法被“反输出（un-emitted）”的；同样，流（stream）的错误状态也会在此时被设置（译注：即产生了错误状态的改变）。在大部分情况下，我们都忽略这些情形；本次讨论的重点是考查「当函数具有两个互不相同的副作用时，如何使函数成为异常安全的」。

[Basic Guarantee vs. Strong Guarantee]

由题可知，该函数满足 basic guarantee：当出现异常的时候，函数不会产生资源泄漏。

该函数不满足 strong guarantee。strong guarantee 意即：如果函数由异常而造成失败，程序的状态必须仍保持不变。然而这里的函数有两个互不相同的副作用（正如函数的名称所暗示的那样）：

1. 一个“…overpaid…”消息被送到 cout；
2. 一个名称字符串被返回。

若考虑到第 2 点，那函数就可以满足 strong guarantee 了，因为当异常产生时，值将不会被返回。若考虑到第 1 点，函数则仍然不是异常安全的，原因有两个：

1. 如果在欲输出消息的第一部分被送到 cout 之后、整个消息被完全送出到 cout 之前的时候有异常被抛出（比如，代码中的第 4 个<<抛出异常），那么此时已经有一部分消息被输出了。[注 1]
2. 如果消息被成功的输出，但在成功输出之后函数产生异常（），那么这个消息也的确已经（无法挽回的）被送到 cout 了，尽管该函数因为一个异常而宣告失败。

要满足 strong guarantee，函数的行为应该是：要么两件事（译注：即输出到 cout 和传值返回）都圆满完成，要么就是遇到该函数抛出异常，两件事都不做。

我们可以达成这样的要求吗？下面是一种我们可能会尝试的方式（不妨称其为第一次尝试）：

```
String EvaluateSalaryAndReturnName( Employee e )
{
  String result = e.First() + " " + e.Last();
  if( e.Title() == "CEO" || e.Salary() > 100000 )
  {
    String message = e.First() + " " + e.Last()
                               + " is overpaid\n";
    cout << message;
  }
  return result;
}
```

这段代码还算不坏。应当注意到，为了让整个 string 只使用一个<<调用，我们用换行符代替了 endl（虽然两者并不完全等同）。（当然，这样做并不能保证「底层的流系统本身不会在对消息施以写操作的时候失败，从而造成不完整的输出」——但我们在这样的高层次已经做了力所能及的努力。）

[一个稍微有点揪心的问题]

到现在，我们仍然有一个微小的瑕疵，它如下面的用户代码（client code）所示：

```
String theName;
theName = EvaluateSalaryAndReturnName( bob );
```

由于函数的结果采用了 return by value(传值返回)方式返回，因此 String 的拷贝构造函数(copy constructor)被唤起；拷贝赋值运算符（copy assignment operator）也被唤起，用来将结果拷贝到 theName。如果这两个拷贝操作中有任一个失败了，那么函数的副作用就已发生效应（因为消息已被完全输出，返回值也已被完全构造好了），而其结果也就无法挽回的丢失了（噢欧!）。

我么能否做得更好一些？可以通过「避免拷贝操作」来避免这个问题吗？这即是说，我们让函数接受一个 non-const 的 String 之引用参数，并将返回值放在这个参数中：

```
void EvaluateSalaryAndReturnName( Employee e, String&  r );
{
  String result = e.First() + " " + e.Last();
  if( e.Title() == "CEO" || e.Salary() > 100000 )
  {
    String message = e.First() + " " + e.Last()
                               + " is overpaid\n";
    cout << message;
  }
  r = result;
}
```

然而不幸的是，对 r 的赋值仍然可能失败，这将造成其中一个副作用被完成而另一个没被完成。最关键的问题在于，这第二次尝试并没有给我们带来多大好处。

于是我们可能会尝试着使用 auto_ptr 来返回结果（不妨把这一次称为第三次尝试）：

```
auto_ptr<String>
EvaluateSalaryAndReturnName( Employee e );
{
  auto_ptr<String> result
      = new String( e.First() + " " + e.Last() );
  if( e.Title() == "CEO" || e.Salary() > 100000 )
```

```
    {
      String message = e.First() + " " + e.Last()
                                 + " is overpaid\n";
      cout << message;
    }
    return result;  // rely on transfer of ownership
  }
```

这正是解题的诀窍之所在——我们有效的隐藏了产生第二个副作用（返回值）的操作，同时也保证了「在第一个副作用（打印消息）被完成后，只使用不抛出异常的（nonthrowing）操作把结果安全的返回给函数调用者」。那么这样做的代价呢？正如在实现强异常安全性（strong exception safety）时经常发生的那样，这种强安全性以效率为代价——我们使用了额外的动态内存分配。

[异常安全性和多重副作用]

从本次讨论可以看出，在第三次尝试中有可能以基本的 commit-or-rollback 语义来完成那两个副作用（与流有关的那个除外）。究其原因，是因为这两个副作用看起来应该可以通过某种技术而被自动完成——这即是说，为两个副作用所做的全部“真正的”工作能够以这样一种方式被完成：即可见的副作用能够只通过不抛出异常的（nonthrowing）操作来完成。

尽管这一次我们还算比较幸运，但情况并不总是那么简单：要编写强异常安全的函数，且让该函数包含两个或多个能被自动完成的、互不相关的副作用（例如，当两个副作用中一个向 cout 送消息，另一个向 cerr 送消息，那会怎么样呢？）——这是不可能的，因为 strong guarantee 要求在出现异常时“程序的状态保持不变”；换句话说，意即只要有异常出现就不能有副作用产生。通常当你遇到两个副作用无法被自动完成的情况时，要实现强异常安全性的唯一方法就是把函数分成两个能自动完成副作用的函数。

本期 GotW 意在描述 3 个重点：

1. 要对强异常安全性提供保证，经常（但并不总是）需要你以放弃一部分性能为代码。
2. 如果一个函数含有多重的副作用，那么其总是无法成为强异常安全的。此时，唯一的方法就是将函数分为几个函数，以使得每一个分出来的函数之副作用能被自动完成。
3. 并不是所有函数都需要具有强异常安全性。本条款中的原始代码和第一次尝试的代码已经能够满足 basic guarantee 了。在许多情况下，第一次尝试的代码已经足够好用，能够将副作用在异常情况下发生的可能性减到最小，而并不需要像第三次尝试那样非要损失一定的性能。

[又及：流和副作用]

正如本条款所示，我们对「被调用的函数没有副作用」之假设并不完全是真实的情况。特别的，我们根本无法保证「流在输出一部分结果之后不会突然失败」。这意味着，我们无法在执行流输出的函数中实现真正的 commit-or-rollback 语义——至少在这些标准流中是不可能的。另外还有一点是，如果流输出失败了，流的状态也将会改变。目前我们不去检查这种情况，也不尝试对其予以恢复——但我们仍可以对函数进行修改，以使其能够捕获由于流而引起的异常，并在重新向调用者抛出异常之前重置 cout 的 error flags。

[注 1]：如果你觉得「担心一条消息是否能够被完全施以 cout 操作」这样的事情多少有点过分学究的味道，那么可以说你的想法并不错。在这里，可能没有人会关心这个。然而任何试图完成两个副作用的函数都是遵循着同样的原理——这也就是为什么我们后续的讨论还有意义的原因。

6. Class 的设计与继承（Class Design and Inheritance）

条款 20：GotW#4 CLASS 技术（Class Mechanics）

难度：7

你对於 classes 的撰写细节知道多少？本条款不仅专注於露骨的错误，也介绍专业程式风格。了解这些原则将有助於你设计出更强固稳健并且更容易维护的 classes。

假设你正在检阅程式码。有位程式员写下这样一个 class，其中有一些不好的写码风格，还有一些错误。你可以找出多少个？有能力修正它们吗？

```
class Complex
{
public:
    Complex( double real, double imaginary = 0 )
        : _real(real), _imaginary(imaginary)
    { }
    void operator+ ( Complex other )
    {
        _real = _real + other._real;
        _imaginary = _imaginary + other._imaginary;
    }
```

```
    void operator<<( ostream os )
    {
        os << "(" << _real << "," << _imaginary << ")";
    }
    Complex operator++()
    {
        ++_real;
        return *this;
    }
    Complex operator++( int )
    {
        Complex temp = *this;
        ++_real;
        return temp;
    }
private:
    double _real, _imaginary;
};
```

解答

这个 class 有许多问题— 甚至比我将明白告诉你的还多。此题重点主要在强调 class 的写作技术（像是「operator<< 的标准型式是什么？」以及「operator+ 应该成为一个 member 吗？」之类的问题），而不在强调其介面设计是多么贫乏。

不过我还是要以一个或许最有用的评语启开序幕：既然标准程式库中已经存在一个 complex class，我们何必再写一个 Complex class? 标准程式库的那个版本并不为以下各问题所苦，它是由工业界最顶尖的人选花费数年完成的。在标准程式库面前请保持谦逊，并尽量重复运用之。

设计准则：尽量重复运用程式码— 特别是标准程式库— 而不要老想著自行撰写。这样不但比较快，也比较容易，比较安全。

或许， 更正上述 Complex 码的最佳办法就是避免使用这个 class ， 改用 std::complex template。

话虽这么说，还是让我们仔细看过整个 class 并修正其中错误吧。首先是 constructor：

1. 以下这个 constructor 允许发生隐式转换（implicit conversion）。

```
        Complex( double real, double imaginary = 0 )
            : _real(real), _imaginary(imaginary)
        { }
```

由於第二个参数有预设值，此函式可被视为单一参数的 constructor，并因此得以允许一个 double 转换为一个 Complex。在本例中这也许是好的，但一如我们在条款 6 所见，这样的转换可能并非总在意图之中。一般而言让你的 constructors 成为 explicit 是个好主意，除非你审慎地决定允许隐式转换。（条款 19 对於隐式转换有更多说明）

设计准则：小心隐式转换所带来的隐式暂时物件。避免这东西的一个好办法就是尽可能让 constructors 成为 explicit，并且避免写出转换运算子。

2. operator+ 或许稍稍有点效率不彰。

```
    void operator+ ( Complex other )
    {
        _real = _real + other._real;
        _imaginary = _imaginary + other._imaginary;
    }
```

为了高效率，参数应该以 by reference to const 的方式传递。

设计准则：尽量以 by const&（而非 by value）的方式来传递物件。

此外，在上述两行中，"a=a+b" 应该重新写为"a+=b"。这么做并不会带给你大量的效率提升（在本例中），因为我们只是对 doubles 进行加法而已。如果是对 class 型别进行加法，效率的改善可能十分明显。

设计准则：尽量写"a op= b;" 而不要写成"a = a op b;"（其中 op 代表任何运算子）。这样不但比较清楚，通常也比较有效率。

为什么 operator+= 比较有效率？因为它直接作用於左侧物件上，并且传回的只是一个 reference，不是一个暂时物件。至於 operator+ 则必须传回一个暂时物件。要知道为什么，请考虑以下标准型式的 operator+= 和 operator+，操作的

对象型别是 T：

```
    T& T::operator+=( const T& other )
    {
        //...
        return *this;
    }
    const T operator+( const T& a, const T& b )
```

```
{
    T temp( a );
    temp += b;
    return temp;
}
```

注意运算子+ 和+= 的关系。前者应该以後者为基础实作出来，这样不但可以简化工作（程式码比较容易撰写），也可以符合一致性（做相同事情的这两个函式不会日後因维护而分道扬镳）。

设计准则：如果你提供了某个运算子的标准版（例如 operator+），请同时为它提供一份 assignment 版（例如 operator+=）并且以後者为基础来实作前者。同时也请总是保存 op 和 op= 之间的自然关系（其中 op 代表任何运算子）。

3. operator+ 不应该是个 member function。

```
void operator+( Complex other )
{
    _real = _real + other._real;
    _imaginary = _imaginary + other._imaginary;
}
```

如果 operator+ 是个 member function，一如本例所示，那么当你决定允许其他型别隐式转换为 Complex 时，operator+ 可能无法以很自然的形式工作。具体地说，当你欲对 Complex objects 加上数值，只能写"a = b + 1.0"而无法写为"a = 1.0 + b"，因为 member operator+ 要求以一个 Complex（而非一个 const double）做为其左侧引数。如果你希望你的使用者能够很方便地为 Complex objects 加上 doubles，提供两个多载化函式：operator+(const Complex&, double) 和 operator+(double, const Complex&) 是合理的想法。

程式准则：使用以下准则来决定一个运算子应该是 member function 或应该是个 nonmember function：

- 一元运算子应该是 members。
- = () [] 和-> 必须是 members。
- assignment 版的运算子（+= -= /= *= 等等）都必须是 members。
- 其他所有二元运算子都应该是 nonmembers。

4. operator+ 不应该修改 this 物件值。它应该传回一个暂时物件，内含相加总和。

```
void operator+ ( Complex other )
{
    _real = _real + other._real;
    _imaginary = _imaginary + other._imaginary;
}
```

注意暂时物件的传回型别应该是"const Complex"（而非仅是"Complex"），以避免这样的使用"a+b=c"。

5. 如果你定义 op，也应该定义 op=。本例之中你应该定义出 operator+=，因为你定义了 operator+，并应该以前者为基础实作出後者。

6. operator<< 不应该成为一个 member function。

```
void operator<<( ostream os )
{
    os << "(" << _real << "," << _imaginary << ")";
}
```

请再看一次 operator+ 的类似讨论。此外，参数应该是"(ostream&, const Complex&)"。注意，nonmember operator<< 通常应该以一个 member function（往往是 virtual）为实作 基础，後者负责真正的工作，常取名为 Print()。

此外，对於一个真正的 operator<<，你应该做某些事情，像是检查 stream 的目前格式旗标（current format flags）以满足常见用法。关於这一部份，请查阅你最喜欢的一本关於标准程式库或 iostreams 的书籍。

7. 更深一层，operator<< 的传回型别应该是"ostream&" 并应该传回一个 reference，代表 stream，以便准许串链式（chaining）输出动作。运用此种方式，使用者便可在程式中以极自然的方式像"cout << a << b;" 这样地使用你的 operator<<。

设计准则：总是在 operator<< 和 operator>> 函式中传回 stream references。

8. 前置式累加运算子的传回型别有误。

```
Complex operator++()
{
    ++_real;
    return *this;
}
```

前置式累加运算子应该传回一个 reference to non-const — 本例而言应该是 Complex&。这可以使程式码的操作更直觉，并避免不必要的效率耗损。

9. 後置式累加运算子的传回型别有误。

```
Complex operator++( int )
{
    Complex temp = *this;
    ++_real;
```

```
        return temp;
    }
```

後置式累加运算子应该传回一个 const 值— 本例而言应该是 const Complex。一旦不允许对传回的物件加以改变，我们便可阻止像"a++++" 这类有问题的码，因为那样的动作结果并不如一个天真的使用者所想像。

10. 後置式累加运算子应该以前置式累加运算子为本。条款 6 有介绍後置式累加运算子的标准型式。

程式准则：为了一致性，请总是以前置式累加运算子为本，实作出後置式累加运算子。否则你的使用者会很惊讶其结果，并往往不高兴。

11. 避免误触保留名称

```
private:
    double _real, _imaginary;
```

是的，十分普及的书籍如 Design Patterns（Gamma95）的确在变数名称身上加了前导底线，但是请你不要这么做。因为标准规格书中保留了某些以「前导底线」开头的识别符号给编译器使用，其中的规则难以记住— 对你以及对编译器撰写者而言，所以你最好完全避免前导底线[注 1]。我比较喜欢的命名法则是为 member 变数名称加上一个後附底线。

（注 1： 举个例子，含有前导底线的名称，技术上只为 nonmember 名称而保留。於是有人主张说 class member 的名称加上前导底线并不会造成问题。事实上这并不完全为真，因为某些编译器以前导底线的方式定义（#define）巨集，而巨集并不重视所谓的 scope。所以它们可以像对你的 nonmember 名称一样，轻易地骴触你的 member 名称。不要使用任何前导底线，或许可以避免这些伤脑筋的事。）

这就是全部的讨论。以下是一份修正後的版本，其中并未涵盖先前没有明白指出的设计和程式风格。

```
class Complex
{
public:
    explicit Complex( double real, double imaginary = 0 )
        : real_(real), imaginary_(imaginary)
    { }
    Complex& operator+=( const Complex& other )
    {
        real_ += other.real_;
        imaginary_ += other.imaginary_;
        return *this;
    }
    Complex& operator++()
    {
        ++real_;
        return *this;
    }
    const Complex operator++( int )
    {
        Complex temp( *this );
        ++*this;
        return temp;
    }
    ostream& Print( ostream& os ) const
    {
        return os << "(" << real_ << "," << imaginary_ << ")";
    }
private:
    double real_, imaginary_;
};

const Complex operator+( const Complex& lhs, const Complex& rhs )
{
    Complex ret( lhs );
    ret += rhs;
    return ret;
}

ostream& operator<<( ostream& os, const Complex& c )
{
    return c.Print(os);
}
```

条款 21：GotW#5 改写虚拟函式（Overriding Virtual Functions）

难度：6

虚拟函式是一个十分基本的性质，但是它们偶尔会暗藏机关，使轻率的你落入圈套之中。如果你能够回答下面这个问题，那么你可以说是相当扎实地了解了虚拟函式，也就比较不会浪费大把时间在相关议题的除错上面。

对著办公室中布满灰尘的程式码档案夹做一番大扫除之後，你看到下面这个不知是谁写的程式片段。写的人似乎对 C++ 的特质有相当的认识。那么，这个程式员期望列印出什么结果呢？实际结果又如何呢？

```
#include <iostream>
#include <complex>
using namespace std;
class Base
{
public:
    virtual void f( int );
    virtual void f( double );
    virtual void g( int i = 10);
};
void Base::f( int )
{
    cout << "Base::f(int)" << endl;
}
void Base::f( double )
{
    cout << "Base::f(double)" << endl;
}
void Base::g( int i )
{
    cout << i << endl;
}
class Derived : public Base
{
public:
    void f( complex<double> );
    void g( int i = 20 );
};
void Derived::f( complex<double> )
{
    cout << "Derived::f(complex)" << endl;
}
void Derived::g( int i )
{
    cout << "Derived::g() " << i << endl;
}
void main()
{
    Base b;
    Derived d;
    Base* pb = new Derived;
    b.f(1.0);
    d.f(1.0);
    pb->f(1.0);
    b.g();
    d.g();
    pb->g();
    delete pb;
}
```

解答

首先，让我们考虑某些风格问题，以及一个真正错误。

1. "void main()" 不是标准写法，因此没有太多移植性。

```
void main()
```

啊呀，是的，我知道这样的写法出现在许多书上。有些作者甚至主张"void main()"可视为近似标准。不，不是这样，即使在 1970s 年代，在尚未有 C++ 标准规格的年代里，这种说法也是不正确的。虽然"void main()" 不是

main 的一个合法宣告，但许多编译器都接受它。不过即使你的编译器接受了"void main()"，并不就代表你可以将它移植到另一个编译器上。因此你最好习惯使用以下两个标准写法之一：

```
int main()
int main( int argc, char* argv[] )
```

你也可能注意到，这个程式并没有在 main 之中写出 return 述句。虽然标准的 main() 应该传回一个 int，但没有写 return 并不会造成问题，因为如果 main()之中没有 return 述句，便是代表"return 0;"。但是我要警告你：在我下笔此刻，还是有许多编译器没有实作出这个规则，因此如果你未明白地於 main() 之中提供一个 return 述句，它们会吐一个警告讯息给你。

2. "delete pb;" 是不安全的。

```
delete pb;
```

这看起来无害。事实上也无害，前提是 Base 得有一个虚拟解构式。本例删除的是一个 pointer-to-base，而该 base class 没有虚拟解构式，这是一种有害行为。你所能预期的最好结果就是程式向下沉沦，因为被唤起的解构式并不正确，而且 operator delete() 会被唤起，接受错误的物件大小。

设计准则：让 base class 的解构式成为 virtual（除非你确定不会有人企图透过一个 pointer-to-base 去删除一个 derived object）。

下面三点很重要，用来区分三个常见的术语：

- 所谓 overload（多载化）一个函式 f()，意思是在同一个 scope 内提供另一个同名（但有不同的参数型别）的函式。当 f() 被呼叫，编译器会根据实际提供的引数型别，设法选择最吻合的一个。
- 所谓 override（改写）一个虚拟函式 f()，意思是在 derived class 内提供一个名称相同，参数型别也相同的函式。
- 所谓 hide（遮掩）一个函式 f()，意思是在一个外围 scope（如 base class 或 outer class 或 namespace）的内层 scope（如 derived class 或 nested class 或 namespace）中提供一个同名函式。它会将外围 scope 的那个同名函式遮掩掉（使它不可见）。

3. Derived::f 不会造成多载化（overload）。

```
void Derived::f( complex<double> )
```

Derived 并不会将 Base::f 多载化，而是会遮掩掉後者。其间区别十分重要，因为这意味 Base::f(int) 和 Base::f(double) 在 Derived 的 scope 中都不可见（令人惊讶的是，某些普及度颇高的编译器面对此等行为，连吐个警告讯息都没有，所以你得完全靠自己的了解）。

如果 Derived 的作者意图遮掩 Base 的函式 f()，那么他如愿以偿[注 2]。然而通常这样的遮掩行为是不经意下的结果，是会令人大吃一惊的。如果你坚持要把名称带进 Derived scope 内，作法是写一个 using declaration "using Base::f;"。

（注 2： 如果要谨慎而从容地遮掩 base class 内的某个名称，比较乾净的作法是为该名称写一个 private using declaration。）

设计准则：当你要提供一个函式，其名称与继承而来的函式同名时，如果你不想因此遮掩了继承而来的函式，请以 using declaration 来突围。

4. Derived::g 改写了 Base::g，但也改变了预设参数。

```
void g( int i = 20 )
```

这绝对是一种不友善的写法。除非你真的想要迷惑众人，否则不要在你改写的函式中改变预设参数值。一般而言「改变预设参数值」并不见得是坏主意，但其用途与作法完全视你个人而定。是的，本例作法在 C++ 是合法的；是的，其结果有良好的定义；但是，噢不，请不要这么做。等一下我们便可以看到它是如何地惑乱我们的思考。

设计准则：绝不要在改写虚拟函式的过程中改变预设参数。

现在我们有了这些不寻常的主题，让我们回头看看程式码，并看看它的行为是否符合你的预期。

```
void main()
{
    Base b;
    Derived d;
    Base* pb = new Derived;
    b.f(1.0);
```

以上都没问题。第一次呼叫唤起的是 Base::f(double)。一如预期。

```
    d.f(1.0);
```

这次呼叫的是 Derived::f(complex<double>)。为什么呢？唔，还记得吗，Derived 并未宣告"using Base::f;"以求将 Base 函式中的 f 带进 scope 内，所以很显然 Base::f(int) 和 Base::f(double) 不能被呼叫。它们并不存在於 Derived::f(complex<double>) 所在的那个 scope 内，因而无法参与多载（overloading）机制。

你可能会以为呼叫的是 Base::f(double)，事实并非如此。甚至也不会引起编译错误，因为很幸运（或说不幸）地，complex<double> 提供了一个隐式转换函式，可将一个 double 转为一个 complex，所以编译器会把这个呼叫动作解释为

```
    Derived::f( complex<double>(1.0) )。
    pb->f(1.0);
```

有趣的是，即使 Base* pb 实际指向一个 Derived object，上一行呼叫的却是 Base::f(double)，因为多载化决议程序(overload resolution)是根据静态型别(static type，本例为 Base)完成，而不是根据动态型别(dynamic type，本例为 Derived) 完成。

基於相同的理由，如果你呼叫 pb->f(complex<double>(1.0)); 将无法通过编译，因为在 Base 介面中没有满足

此一呼叫的函式。

```
    b.g();
```

这会印出"10"，因为它很单纯地唤起 Base::g(int)，而其预设参数为 10。简单！

```
    d.g();
```

这会印出"Derived::g() 20"，因为它很单纯地唤起 Derived::g(int)，而其预设参数为 20。简单！

```
    pb->g();
```

这会印出"Derived::g() 10"。

『等等』，你抗议道，『发生了什么事？』这个结果可能会短暂地锁死你的神智，将你带往一个歇斯底里的状态，直到你终於了解编译器的所做所为完全适当[注 3]。记住一件事，和 overload（多载化）一样，所谓预设参数，系根据物件的静态型别（static type，本例为 Base）来决定，因此此时的预设参数为 10。然而由於 g() 是个虚拟函式，所以被唤起的函式是根据动态型别（dynamic type，本例为 Derived）来决定。

（注 3 虽然，当然啦，Derived 的设计者应该被拖到停车场接受大家的嘲笑。）

如果你真正了解了上述关於「名称遮掩」以及「何时使用静态型别，何时使用动态型别」的叙述，你才算是真正很酷地了解了这个题目。恭喜你！

```
    delete pb;
}
```

最後，一如先前要你注意的，上述删除动作会腐蚀你的记忆体，导至局部败坏。

请看先前对虚拟解构式（virtual destructor）的讨论。

条款 22：GotW#14 Classes 之间的关系（Class Relationships）（一）

难度：5

你的 OO 技巧如何？本条款举例说明一个 class 设计上的常见错误，此错误至今仍然对许多程式员构成陷阱。

有一个网路应用程式，拥有两种通讯行为，每一种行为有自己的讯息协定。两个协定之间有一些相似性（某些运算、乃至某些讯息相同），所以程式员完成下列设计，将共同的运算和讯息封装於一个 BasicProtocol class 内。

```
class BasicProtocol /* : possible base classes */
{
public:
    BasicProtocol();
    virtual ~BasicProtocol();
    bool BasicMsgA( /*...*/ );
    bool BasicMsgB( /*...*/ );
    bool BasicMsgC( /*...*/ );
};
class Protocol1 : public BasicProtocol
{
public:
    Protocol1();
    ~Protocol1();
    bool DoMsg1( /*...*/ );
    bool DoMsg2( /*...*/ );
    bool DoMsg3( /*...*/ );
    bool DoMsg4( /*...*/ );
};
class Protocol2 : public BasicProtocol
{
public:
    Protocol2();
    ~Protocol2();
    bool DoMsg1( /*...*/ );
    bool DoMsg2( /*...*/ );
    bool DoMsg3( /*...*/ );
    bool DoMsg4( /*...*/ );
    bool DoMsg5( /*...*/ );
};
```

每一个 DoMsg…() member function 都呼叫 BasicProtocol::Basic…() 执行共同的工作，然後 DoMsg…() 另外再执行其实际传送工作。每一个 class 都还可以有其他的 members，但你可以假想所有重要的 members 都已呈现出来。

请评论此一设计。有任何东西需要改变吗？如果有，为什么？

解答

这个条款以实例说明一个十分常见的 OO classes 相互关系上的错误设计。让我再说一次重点，classes

Protocol1 和 Protocol2 以 public 方式继承了一个共同的 base class BasicProtocol，後者执行某些共同的工作。

问题的关键在於以下这段叙述：

每一个 DoMsg…() member function 都呼叫 BasicProtocol::Basic…() 执行共同的工作，然後 DoMsg…() 另外再执行其实际的传送工作。

抓到它了：这很明显是在描述一个「以某物为基础实作出」（is implemented in terms of）的关系，这种关系在 C++ 中应该以 private inheritance 或 membership 来完成。

不幸的是许多人往往误用 public inheritance，因而将 implementation inheritance（实作继承）和 interface inheritance（介面继承）混淆在一起。两者并非相同的东西，而如此这般的混淆正是此处问题的根源。

常见错误：绝不要使用 public inheritance，除非你要模塑的是真正的 Liskov IS-A 和 WORKS-LIKE-A 的关系。所有被改写（overridden）的 member functions 不能要求更多也不能承诺更少。

附带一提，程式员如果习惯制造这种错误（以 public inheritance 方式来进行实作继承而非介面继承），往往会导致深度的继承体系。造成不必要的复杂度，因而大大增加维护上的负荷，并迫使使用者学习许多 classes 介面 — 纵使他们所要的或许只是使用某个特定的 derived class 而已。这种错误也可能对记忆体的使用和程式的效率造成冲击，因为非必要的 vtables 以及非必要的「classes 间接性」都增加了。如果你发现自己常常产生深度继承体系，应该反省你的设计风格，看看是否染上了这个坏习惯。深度继承体系很少有其必要，而且几乎绝对不理想。如果你不同意我的话，如果你认为「没有很多继承的 OO，不是 OO」，那么请你想想 C++ 标准程式库。

设计准则：绝对不要以 public inheritance 复用（reuse）base class 内的程式码；public inheritance 是为了被复用（reused）— 被那些「以多型方式运用 base objects」的程式码复用（注 4）。

（注 4：我从 Marshall Cline 获得这个设计准则，并深深感谢。Cline 是经典作品 C++ FAQs 的作者之一（Cline 95）。）

下面是更详细的说明，有数条线索帮助我们指出问题所在。

1. BasicProtocol 没有提供任何虚拟函式（以及 destructor，此点稍後再论）（注 5）。这意味它并不意图以多型的方式（polymorphically）被使用。这对 public inheritance 是一个强烈的反对暗示。

（注 5：纵使 BasicProtocol 本身系衍生自另一个 class，我们的结论还是相同，因为它还是不提供任何「新的」虚拟函式。如果它的 base class 确实提供了虚拟函式，意思是这个更远端的 base class（而不是 BasicProtocol 自己）意图以多型的方式被使用。所以，如有必要，我们应该由那个更远端的 base class 直接衍生下来。）

2. BasicProtocol 没有任何 protected members 。这意味没有任何衍生介面（derivation interface），这对任何继承型式（不论 public 或 private），都是一个强烈的反对暗示。
3. asicProtocol 封装了共同的工作，但是一如先前所述，它并不像 derived classes 那样执行自己的传送动作。这意味 BasicProtocol 物件运作起来并非「像一个（WORK-LIKE-A）」衍生的 protocol 物件，也并非「使用起来可视为一个（USABLE-AS-A）」衍生的 protocol 物件。Public inheritance 应该只能用来模塑唯一一件事情：一个真正的「IS-A 介面关系」，奉行「Liskov 替换原则」（Liskov Substitution Principle）。为了让文意更清晰，我常称此替换原则为 WORKS-LIKE-A 或 USABLE-AS-A[注 6]。

（注 6：是的，有时候，当你以 public 方式继承了一个介面，某些实作细节会跟著一并过来—如果 base class 不但拥有你需要的介面，还拥有它自己的实作细目（译注：意指拥有其 data members）的话。我们当然可以解决这样的问题（见下一条款），但是要维护这么纯粹的「一个 class 一份责任」，也似乎并非总有必要。）

4. 所有衍生下去的 classes 只使用 BasicProtocol 的 public 介面。这意味它们并没有因为自己是 derived classes 而受益；它们的工作可轻易以一个型别为 BasicProtocol 的辅助物件来完成。

这意思是，我们在清理方面有了一些功课要作做。首先，由於 BasicProtocol 很明显地并非被设计用来让其他 classes 继承，所以其 virtual destructor 没有必要（如果存在反而会误导），应该去除。其次，BasicProtocol 或许应该重新命名（例如 MessageCreator 或 MessageHelper），以避免误会。

一旦我们完成这些改变，接下来应该决定以哪一种方法来模塑"is implemented in terms of" 关系。使用 private inheritance 或 membership？答案很容易记住。

设计准则：当我们希望模塑出"is implemented in terms of" 的关系，请选择 membership/aggregation 而不要使用 inheritance。只有在绝对必要的情况下才使用 private inheritance — 也就是说当你需要存取 protected members 或是需要改写虚拟函式时。绝对不要只为了重复运用程式码而使用 public inheritance。

如果使用 membership，可以强迫事情有比较好的区分，因为使用者（class）是个一般的 client 端，只能取用被使用者（也是个 class）的公开介面。采用它，你会发现你的程式码比较乾净，比较容易阅读，也比较容易维护。简单地说就是，你的程式码成本降低了。

条款 23：GotW#15 Classes 之间的关系（Class Relationships）（二）

难度：6

Design patterns 是一个重要的工具，用来撰写可重用的码。你可以辨识出本条款所使用的 patterns 吗？如果是，你可以改善它们吗？

资料库应用软体往往需要对某个已知表格内的每一笔记录（record）或是一群圈选起来的记录（selected records）做某些动作；它们常常会先执行一次唯读动作，将整个表格走访一遍，放进快取装置（cache）中，以便获得「哪些记录需要被处理」的资讯，然後再进入第二回合，做实际的改变动作。

程式员不希望一再重复撰写这种常见的逻辑，於是试著在以下抽象类别中提供一个泛型而可复用的框架（generic reusable framework）。其想法是，首先，抽象类别应该将重复工作封装起来，收集需要操作的各个表格行（table rows）；其次，再对每一行执行必要的动作。衍生类别负责提供特定行为的细节。

//----------

```
// File gta.h
//-----------------------------------------------
class GenericTableAlgorithm
{
public:
    GenericTableAlgorithm( const string& table );
    virtual ~GenericTableAlgorithm();
    // Process() 於成功时传回 true。
    // 该函式负责所有工作：a) 实际读取表格中的记录（records），
    // 并为每一笔记录呼叫 Filter() 以决定是否应该被放置於
    // 「待被处理的 rows」之中；b) 当「待被处理的 rows」
    // 所组成的 list 架构完毕後，为每一个 row 呼叫 ProcessRow()
    //
    bool Process();
private:
    // 如果接获的记录（record）应该被放进「待被处理的 rows」之中，
    // Filter() 就传回 true。预设行为是传回 true（以便含括所有的 rows）。
    //
    virtual bool Filter( const Record& );
    // 每当一笔记录（record）被含括以便更进一步处理时，
    // ProcessRow() 会被呼叫。这是具象类别进行其特定
    // 工作的地方。(注意：这意味被处理的每一个 row 都
    // 将被读取两次，但是我们假设这是必要的，不构成效率
    // 上的考量）
//
    virtual bool ProcessRow( const PrimaryKey& ) =0;
    class GenericTableAlgorithmImpl* pimpl_; // MYOB
};
```

举个例子，client 端衍生了一个具象的 class，并在 main() 之中这样使用它：

```
class MyAlgorithm : public GenericTableAlgorithm
{
    // ... 改写（override）Filter() 和 ProcessRow() 的行为
    // 实作出特定动作...
};
int main()
{
    MyAlgorithm a( "Customer" );
    a.Process();
}
```

我的问题是：

1. 这是个好设计，实作出一个广为人知的 design pattern。哪个 pattern？为什么它在此处有用？
2. 在不改变基础设计的情况下，试评论此一设计的执行方式。你可能会有什么不同的作法？pimpl_ member 的目的是什么？
3. 这个设计事实上有改善空间。什么是 GenericTableAlgorithm 的任务？如果其任务超过一个，它们如何可以被更好地加以封装？试说明你的答案如何影响这个 class 的重用性，尤其是其扩充性。

解答

让我们一个一个来。

1. 这是个好设计，实作出一个广为人知的 design pattern。哪个 pattern？为什么它在此处有用？

这是 Template Method（Gamma95）pattern（请不要与 C++ templates 搞混了）。它之所以有用，因为我们只需遵循相同步骤，就可以将某个常见解法一般化。只有细节部份可能不同，而此部份可由衍生类别供应。甚且，一个衍生类别也可能选择再次施行 Template Method —— 也就是说它可以将虚拟函式改写（override）为另一个虚拟函式的外包函式（wrapper）—— 於是不同的步骤就可以被填入 class 阶层体系的不同层级中。

（注意：Pimpl 手法表面上类似 Bridge，但此处仅只用来遮掩 class 自己的实作细节，作用类似编译依存性（compilation dependency）的防火墙，不是用来当做一个真正的可扩充的 bridge。我将在条款 26~30 深度分析 Pimpl 手法）。

设计准则：尽量避免使用 public 虚拟函式；最好以 Template Method pattern 取代之。

设计准则：了解什么是 design patterns，并运用之。

2. 在不改变基础设计的情况下，试评论此一设计的执行方式。你可能会有什么不同的作法？pimpl_ member 的目的是什么？

这个设计以 bools 做为传回码，显然没有其他方法（诸如状态码或 exceptions）可用来记录失败。视需求而定，

这或许是好的，但有些东西需要注意。程式中的 pimpl_ member 非常巧妙地将实作细节隐藏於一个不透明指标之後。pimpl_ 指向的结构将内含 private member functions 和 member variables，使得它们的任何改变都不至於造成 client 端有必要重新编译。这是一个由 Lakos（Lakos96）及其他人提出的重要技术，它虽然对写码造成了一点困扰，但的确补偿了「C++ 缺乏模组系统（module system)」的遗憾。

设计准则：为了广泛运用 classes，最好使用「编译器防火墙」手法（compiler-firewall idiom，或称为 Pimpl Idiom）来遮掩实作细节。使用一个不透明指标（此指标指向一个已宣告而未定义的 class），将之宣告为"struct XxxxImpl* pimpl_;"，用来存放 private members（包括 state variables 和 member functions）。例如"class Map { private: struct MapImpl* pimpl_; };"。

3. 这个设计事实上有改善空间。什么是 GenericTableAlgorithm 的任务？如果其任务超过一个，它们如何被更好地加以封装？试说明你的答案如何影响这个 class 的复用性，尤其是其扩充性。

我们可以实质改善 GenericTableAlgorithm，因为它同时兼顾两份工作。当你必须兼顾两份工作，你会有压力，因为你的负担过重而且任务之间可能彼此竞争。这个 class 也是一样，所以调整其焦点应该能够带来利益。

在原版本中，GenericTableAlgorithm 承担两个不同且互不相干的任务，可以被有效地隔离，原因是这两份任务支援完全不同的客户。简单地说这两种客户是：

- client 端，使用（经过适当特殊化後的）泛型演算法。
- GenericTableAlgorithm，使用特殊化後的 concrete "details" class 以期针对某特定情况或用途而特殊化其行为。

设计准则：尽量形成内聚（cohesion）。总是尽力让每一段码— 每一个模组、每一个类别、每一个函式— 有单一而明确的任务。

现在，让我们看看改良後的成果：

```
//------------------------------------------------
// File gta.h
//------------------------------------------------
// 任务#1: 提供一个公开介面，用来封装共同机能，
// 使之成为一个 template method。这与继承全然无关，
// 可被巧妙地隔离，使本身成为一个容易集中注意力的 class。
// 客户目标锁定 GenericTableAlgorithm 的外部使用者。
//
class GTAClient;
class GenericTableAlgorithm
{
public:
    // Constructor 如今接受一个具象的 implementation object.
    //
    GenericTableAlgorithm( const string& table,
    GTAClient& worker );
    // 由於我们已经从继承关系中抽离出来，
    // destructor 不再需要为 virtual。
    //
    ~GenericTableAlgorithm();
    bool Process(); // 此行未有改变
private:
    class GenericTableAlgorithmImpl* pimpl_; // MYOB
};
//------------------------------------------------
// File gtaclient.h
//------------------------------------------------
// 任务#2: 提供一个抽象介面，为的是扩展性。
// 这是 GenericTableAlgorithm 的一个实作细节，与其
// 外部 clients 全然无关，可被巧妙地抽离为一个
// 容易集中注意力的 abstract protocol class。
// 客户目标锁定 concrete "implementation detail" classes
// 的撰写者，他们使用（并扩展）GenericTableAlgorithm。
//
class GTAClient
{
public:
    virtual ~GTAClient() =0;
    virtual bool Filter( const Record& );
    virtual bool ProcessRow( const PrimaryKey& ) =0;
};
//------------------------------------------------
```

```
// File gtaclient.cpp
//---------------------------------------------------
bool GTAClient::Filter( const Record& )
{
    return true;
}
```

一如所示，这两个 classes 应该出现在不同的表头档中。有了这些改变，现在 client 端看起来应该如何？答案是：和以前非常近似。

```
class MyWorker : public GTAClient
{
    // ... 改写（override）Filter() 和 ProcessRow() 的行为
    // 实作出特定的动作...
};
int main()
{
    GenericTableAlgorithm a( "Customer", MyWorker() );
    a.Process();
}
```

虽然看起来非常近似，不过请注意三个重要影响。

1. 如果 GenericTableAlgorithm 的公共公开介面改变了（例如加入一个新的 public member），会如何？在原来的设计中，所有具象的 worker classes 都必须重新编译，因为它们衍生自 GenericTableAlgorithm。在新版本中，GenericTableAlgorithm 公开介面的任何改变都被巧妙地隔离，一点也不会影响具象的 worker classes。
2. 如果 GenericTableAlgorithm 的可扩充协定改变了（例如 Filter() 或 ProcessRow() 增加了一些额外的预设引数），会如何？在原来版本中，GenericTableAlgorithm 的所有外部 clients 都必须重新编译（即使公开介面并未改变），因为在此 class 定义区中可见到衍生介面。但是在新版本中，GenericTableAlgorithm 的扩充协定介面被巧妙地隔离，其任何变化都不会影响外部使用者。
3. 任何具象的 worker classes 如今可以在其他任何演算法中被使用— 只要该演算法能够使用 Filter()/ProcessRow() 介面来进行运算— 而不只是被 GenericTableAlgorithm 使用而已。事实上我们最後所得的结果非常近似 Strategy pattern。

记住计算机科学的箴言：大部份问题都可以藉由「多加一层间接性」获得解决。当然啦，以奥卡姆剃刀（Occam's Razor，意指把论题简化的思考原则）做为调节是聪明的作法：不要让事情过度复杂。两者间的平衡可以在此例中形成更好的复用性和维护性，只花极少成本或甚至不需任何成本— 这对任何客户而言都是一笔极佳交易。

让我们多花一点时间谈谈所谓的泛型（genericity） 。你可能已经注意到，GenericTableAlgorithm 其实可以是个函式，不必一定得是 class。事实上，某些人可能会受到诱惑，将 Process() 重新命名为 operator()()，因为现在这个 class 很明显其实只是个 functor（译注：即 function object）而已。它可以被取代为一个函式的原因是，其需求描述中并没有说它需要在各个 Process() 呼叫之间保持状态（state）不变。举个例子，如果它不需要在唤起之间保持状态，我们可以将它改为：

```
   bool GenericTableAlgorithm(
   const string& table,
   GTAClient& method
)
{
// ... Process() 的原本内容...
}
int main()
{
    GenericTableAlgorithm( "Customer", MyWorker() );
}
```

这里我们真正获得的，是个泛型函式（generic function），可以在必要的时候指定特殊化行为。如果你知道 method objects 从不需要储存状态（也就是说其所有实体的机能都相同并且只提供虚拟函式），你可以更精致地以一个「非类别的 template 参数」取代 method。

```
template<typename GTACworker>
bool GenericTableAlgorithm( const string& table )
{
    // ... Process() 的原本内容...
}
int main()
{
    GenericTableAlgorithm<MyWorker>( "Customer" );
}
```

除了 client 端得以省去一个逗点，我并不认为这会使你得到很大的收获。所以第一个函式是比较好的。请拒绝只因为自己的利益而写出一些险招秘技。

无论如何，於某种已知形势下，究竟要使用函式抑或 class，取决於你企图达到什么目标；本例中，撰写泛型函式或许是个比较好的解答。

条款 24：使用/滥用继承（Uses and Abuses of Inheritance）

难度：6

为什么使用继承？

继承往往被过度运用，即使是经验丰富的开发人员也会犯这样的错误。请记住，耦合关系（coupling）要尽量减少。如果 class 与 class 之间的关系可以多种方式表达，请使用其中关系最弱的一种。继承几乎是 C++ 所能表达的最强烈关系（仅次於朋友关系），只有在无「效能相等而关系较弱」的情况下，才适用继承。

本条款焦点在於 private inheritance、一个真正（但可能晦涩）的 protected inheritance 运用实例，以及适度运用 public inheritance 的要点重述。沿此路线，我们将对「继承的运用」做出一份恰当而完整的整理，列出常见和不常见的理由。

以下 template 提供了 list（串列）的管理功能，包括在特定的 list 位置上处理元素的能力。

```
// Example 1
//
template <class T>
class MyList
{
public:
    bool Insert( const T&, size_t index );
    T Access( size_t index ) const;
    size_t Size() const;
private:
    T* buf_;
    size_t bufsize_;
};
```

考虑下面程式码，其中呈现「以 MyList 为基础，实作出一个 MySet class」的不同作法。假设所有重要元素都已呈现。

```
// 例 1(a)
//
template <class T>
class MySet1 : private MyList<T> // 译注：private inheritance
{
public:
    bool Add( const T& ); // calls Insert()
    T Get( size_t index ) const;
    // calls Access()
    using MyList<T>::Size; // 译注：注意这个 using declaration
    //...
};
// 例 1(b)
//
template <class T>
class MySet2
{
public:
    bool Add( const T& ); // calls impl_.Insert()
    T Get( size_t index ) const;
    // calls impl_.Access()
    size_t Size() const; // calls impl_.Size();
    //...
private:
    MyList<T> impl_; // 译注：containment
};
```

请思考以下问题：

1. MySet1 和 MySet2 之间有任何差别吗？
2. 更广泛地说，nonpublic inheritance 和 containment 有什么不同？尽你所能完成一份列表，说明为什么你采用 inheritance 而不采用 containment。
3. 你比较喜欢哪个版本？MySet1 或 MySet2？
4. 尽你所能完成一份列表，说明为什么你选择 public inheritance。

解答

这个刺激的例子有助於解释继承机制使用上的某些议题，特别是如何在 nonpublicinheritance 和 containment 之间做抉择。

第一个问题「MySet1 和 MySet2 之间有任何差别吗？」答案十分简明：两者之间不具有实际意义上的差别。它们的机能完全相同。

第二个问题使我们开始严肃地思考某些东西：更一般地说，nonpublic inheritance 和 containment 有什么不同？尽你所能完成一份列表，说明为什么你采用 inheritance 而不采用 containment。

- Nonpublic inheritance 应该总是表现「根据某物实作出（IS-IMPLEMENTED-IN-TERMS-OF)」的意义（唯一罕见例外将於稍後显现)。它使 using class 得以依赖 used class 的 public/protected 成份。
- Containment 总是表现出「有一个（HAS-A）」的意义，连带也有「根据某物实作出（IS-IMPLEMENTED-IN-TERMS-OF)」的意义。它使 using class 只能依赖 used class 的 public 成份。

很容易看出来，inheritance 是 single containment 的一个超集，也就是说，没有什么是我们可以用一个 MyList<T> member 完成而却无法以「继承自 MyList<T>」完成的事。当然啦，inheritance 的运用造成我们只能拥有一份 MyList<T>（做为 base subobject）；如果我们需要拥有多份 MyList<T> 实体，就必须改用 containment。

设计准则：尽量采用 aggregation（又名"composition", "layering", "HAS-A","delegation"）来取代 inheritance。当我们准备模塑 IS-IMPLEMENTED-IN-TERMS-OF 关系时，尽量采用 aggregation 而不要使用 inheritance。

於是我们想知道，如果采用 inheritance，可以做到什么额外的事情？如果采用 containment，有些什么事是我们做不到的？从另一个角度说，为什么要使用 nonpublic inheritance 呢？下面有一些理由，大致上从最频繁排到最罕见。有趣的是，最後一项理由亦指出 protected inheritance 的一个有价值（?）的应用。

- 当我们需要改写（override）虚拟函式。这是 inheritance 的经典使用理由[注 7]。通常我们之所以希望改写，是为了订制 using class 的行为。然而有时候别无其他选择。如果 used class 是抽象的，也就是说它有至少一个纯虚拟函式尚未被改写，我们就必须继承并改写它，因为我们无法直接将它具现化。
- 当我们需要处理 protected member — 一般而言是 protected member functions[注 8],有时则是指 protected constructors。
- 当我们需要在一个 base subobject 之前先建构 used object，或是在一个 base subobject 之後摧毁 used object。如果如此些微的寿命差异形成程式的重点，我们除了使用 inheritance，别无它法。当 used class 提供某种锁定（lock）机制，例如一个关键区域（critical section）或一笔资料库交易，而它们必须涵盖另一个 base subobject 的整个寿命，恐怕就符合本因素了。

（注 7：请见 Meyers98 索引中的"French, gratuitous use of"。）

（注 8：我说 member functions，因为你绝不会写一个 class 并令它拥有 public 或 protected member variable，对不？（不要管某些程式库提供的低劣例子））

- 当我们需要(1) 分享某个共同的 virtual base class 或(2) 改写某个 virtual base class 的建构程序。如果 using class 必须继承相同的 virtual bases 之一做为 used class，(1) 部份可以适用。如果不是这样，(2) 部份仍然适用。衍生程度最深（most-derived）的 class 有责任初始化所有的 virtual base classes，所以如果我们需要针对一个 virtual base 使用不同的 constructor 或不同的 constructor 参数，那么我们就必须使用 inheritance。
- 当我们从 empty base class 的最佳化获得实质利益。有时候，你据以实作的那个 class 可能并没有任何 data members — 换句话说它只是函式的组合。这种情况下以 inheritance 取代 containment，可因 empty base class 最佳化之故而获得空间优势。简单地说，编译器允许一个 empty base subobject 占用零空间；虽然，一个 empty member object 必须占用非零空间— 即使它不含任何资料。

```
class B { /* ... 只有函式，没有资料... */ };
// Containment: 招致某种空间上的额外负担
//
class D
{
    B b_; // b_ 必须占用至少一个 byte，
}; // 即使 B 是一个空的 class
// Inheritance：不会带来空间上的额外负担
//
class D : private B
{ // B base subobject 不需要
}; // 占用任何空间
```

关於 empty base 最佳化的详细讨论，Nathan Myers 有一篇好文章，发表於 Dr.Dobb's Journal（Myers97)。

让我对这种过度热心的行为加上一个警告：不是所有编译器都能够完成 empty baseclass 的最佳化。就算它们有这个能力，你所获得的利益也不一定有意义，除非你知道你的系统中有许多（例如成千上万个）这样的物件。除非空间的节省对你的应用程式非常重要，而且你知道你的编译器有能力做最佳化，否则，以较强烈的 inheritance 关系取代较微弱的 containment 关系，会导致更多的耦合（couple)，那是不智之举。

另外还有一个性质，我们也可以使用 nonpublic inheritance。这是唯一一个并非模塑 IS-IMPLEMENTED-IN-TERMS-OF 关系者：

- 当我们需要「受控制的多型性（controlled polymorphism)」，也就是 LSP ISA，但只在某些码身上。Liskov SubstitutionPrinciple (LSP) [注 9]告诉我们，public inheritance 总是应该用来模塑 IS-A 的关系。

Nonpublic inheritance 可以表现一个受到束缚的 IS-A 关系，虽然，大部份人只以 public inheritance 来识别 IS-A。

（注 9： 请看 www.oma.com，其中有不少探讨 LSP 的好文章。）

假设有个 class Derived : private Base，从外部看它，Derived 物件不是一个（IS-NOT-A）Base。所以它当然无法以多型方式被当做一个 Base，因为 private inheritance 带来存取上的束缚。然而，在 Derived 自己的 member functions 及 friends 内，一个 Derived object 真的可以以多型方式被拿来当做一个 Base（你可以在需要 Base object 的任何地点提供一个「指向 Derived object」的 pointer 或 reference。这是因为 members 和 friends 有足够的存取权限。如果不采用 private inheritance 而改用 protected inheritance，那么 IS-A 的关系对更深的 derived classes 会更明显些，意味 subclasses 也可以使用多型（polymorphism）。

这就是尽我所能列出的一份列表，详载 nonpublic inheritance 的使用理由。事实上只要再加一点：我们需要 public inheritance 以表现 IS-A，这份列表就对任何种类之 inheritance 的使用有了详细的理由。问题 4 对此会有更多讨论。所有这些东西把我们带到问题 3：你比较喜欢 MySet 的哪个版本？MySet1 还是 MySet2？让我们分析例子 1 中的程式码，看看是否以上任何一个标准都适用。

- MyList 没有 protected members，所以我们不需要继承它以求存取它们。
- MyList 没有虚拟函式(s)，所以我们不需要继承它以求改写它们。
- MySet 没有其他潜在的 base classes，所以 MyList 物件不需要在另一个 basesubobject 之前建构或之後摧毁。
- MyList 并没有任何 virtual base classes 是(1) MySet 可能需要共享的，或(2)其 construction 可能需要改写的。
- MyList 不是空的，所以「empty base class 最佳化」的动机并不适用。
- MySet 不是一个（IS-NOT-A）MyList，甚至在 MySet 的 member functions 和 friends 内也不是。最後一点很有趣，因为它指出 inheritance 的一个极小缺点。即使其他评断条件中的某一个是真的，使我们决定使用 inheritance，我们还是得小心，不让 MySet 的 members 和 friends 出乎意料地以多型方式将一个 MySet 视为一个 MyList — 或许这只是个遥远的可能，但如果真的发生，却是相当难解，或许会令可怜的程式员迷惑数小时。

简单地说，MySet 不应继承 MyList。在 containment 同样有效率的情况下使用 inheritance，只会导入无偿的耦合及非必要的相依性而已，那绝不是个好主意。不幸的是，在真实世界中，我还是会看到经验丰富的程式员以 inheritance 的方式来实作 MySet。

机敏的读者一定注意到了，「inheritance 版」的 MySet 比起「containment 版」提供了一个（颇为无关痛痒的）优点。使用 inheritance，你只需写一个 usingdeclaration 就可以将未曾更改的 Size 函式曝光。如果使用 containment，你必须明白写出一个简易的转交函式（forwarding function）以获得相同效果。

当然，有时候 inheritance 是比较合适的。例如：

```
// 例 2: Sometimes you need to inherit
//
class Base
{
public:
virtual int Func1();
protected:
bool Func2();
private:
bool Func3(); // uses Func1
};
```

如果我们需要改写一个虚拟函式如 Func1，或存取一个 protected member 如 Func2，就非得采用 inheritance 不可。例 2 说明为什么即使并非为了多型（polymorphism），改写虚拟函式也可能是必要的。在这里，Base 系根据 Func1 实作出来（Func3 在其实作中使用 Func1），所以唯一取得正确行为的方法就是改写 Func1。然而即使 inheritance 是必要的，下面是正确的作法吗？

```
// 例 2(a)
//
class Derived : private Base // 有必要吗？
{
public:
int Func1();
// ... 更多函式，其中某些使用 Base::Func2()，
// 另外某些则不使用...
};
```

这份码允许 Derived 改写 Base::Func1，那很好。不幸的是，它也允许让所有的 Derived members 取用 Base::Func2。或许只有一些（或甚至一个）Derived member functions 真正需要取用 Base::Func2。但是透过上面那样的继承机制，我们却让所有的 Derived members 仰赖 Base 的 protected interface，那是不必要的。

很明显，inheritance 是有必要，但是否能够只导入我们真正需要的耦合就好了呢？

唔，加上一点点工程，我们可以做得更好。

```
// 例 2(b)
```

```
//
class DerivedImpl : private Base
{
public:
int Func1();
// ... 这里的函式需要使用 Func2 ...
};
class Derived
{
// ... 这里的函式不需使用 Func2 ...
private:
DerivedImpl impl_;
};
```

这个设计好多了，因为它良好地区隔并封装了对 Base 的依存性。Derived 只直接依赖 Base 的 public 介面以及 DerivedImpl 的 public 介面。为什么这样的设计比较成功呢？主要是因为它遵循了「一个 class 一个任务」的基本设计准则。在例 2(a) 中，Derived 的任务是同时订制 Base 并以 Base 为基础实作自己。在例 2(b) 中，那些都被良好地分隔开了。

现在我们看看 containment 的情况。Containment 有它自己的某些优点，第一，它允许我们拥有多个 used class 实体，这对 inheritance 而言并不实用，甚至不可能。

如果你既需要衍生又需要多份实体，请使用例 2(b) 所展示的手法：衍生一个辅助类别（如 DerivedImpl）负责继承事务，然後再在这辅助类别中内含多个实体。

第二，它令 used class 成为一个 data member，这带来更多弹性。那个 member 可以在一个 Pimpl10 之中被隐藏於编译器防火墙後面（然而 base class 的定义必须总是可见），而且如果有必要在执行期被改变的话，它甚至可以轻易地被转换为一个指标。(尽管继承体系是静态的并於编译期就固定了)。

下面是第三个有用的方法，重写例 1(b) 中的 MySet2，以更广泛的方式来使用 containment。

```
// 例 1(c): Generic containment
//
template <class T, class Impl = MyList<T> >
class MySet3
{
public:
    bool Add( const T& ); // calls impl_.Insert()
    T Get( size_t index ) const;
    // calls impl_.Access()
    size_t Size() const; // calls impl_.Size();
    // ...
private:
    Impl impl_;
};
```

我们现在有了更多弹性，不再只是以 MyList<T> 为基础完成实作，更可以令 MySet 以任何 class 为基础完成实作— 只要它们支援上述的 Add, Get 以及其他需要用到的函式。C++ 标准程式库就是使用这项技术来完成其 stack 和 queue 两个 templates，两者在预设情况下是以一个 deque 为基础，但它们也允许以任何其他 class 为基础— 只要那些 classes 提供必要服务。

不同的使用者可能会根据不同的效率特性来决定 MySet 如何具现化。举个例子，如果我知道我将进行的 insert 动作远多於 search 动作，我会使用对 insert 动作最佳化的某个实作品。我们并没有失去任何易用性。在例 1(b) 中，client 码可以只简单地写 MySet2<int>，具现出一个 ints 组成的集合，例 1(c) 仍然可以这样，因为 MySet3<int> 只不过是 MySet3<int,MyList<int> > 的同义词而已— 这真得感谢所谓的 default template parameter。

这种弹性比较难以藉由 inheritance 达成，主要因为 inheritance 倾向在设计期就将实作固定下来。是有可能将例 1(c) 改为继承自 Impl 啦，但是太紧绷的耦合关系不但不必要，而且应该避免。

关於 public inheritance，我们可以从问题 4 的答案中学习到最重要的事情。问题 4 如下：尽你所能完成一份列表，说明为什么你使用 public inheritance。

关於 public inheritance，我只强调一个重点。如果你奉行此一忠告，它就会导引你看清最常见的滥用情况。记住，永远只把 public inheritance 用来模塑真正的 IS-A 关系，一如 Liskov Substitution Principle 所说[注 11]。也就是说任何情况下，只要可以使用 base class object，就应该可以使用 publicly derived class object。注意，条款 3 有一个罕见例外（或更正确地说是一个扩充）。

（注 11: 请看 www.oma.com，其中有不少探讨 LSP 的好文章。）

更明确地说，遵循以下规则就可以避免常见的一些易犯错误。

- 如果 nonpublic inheritance 堪用，就绝对不要考虑 public inheritance。Public inheritance 绝不应该在缺乏真正 IS-A 关系的情况下被拿来模塑 ISIMPLEMENTED-IN-TERMS-OF 关系。这似乎显而易见，但是我注意到某些程式员常态性地习惯运用 public inheritance。那不是好习惯。这一点和我的另一个忠告的精神一样：当好的旧的 containment/membership 还堪用时，绝不要使用 inheritance。如果不需要额外的存取能力或是更紧绷的耦合性，何必使用它呢？

- 如果 class 相互关系可以多种方式来表现，请使用其中最弱的一种关系。 绝对不要以 public inheritance 来实作"IS-ALMOST-A"的关系。我曾经看过某些程式员，甚至是经验丰富的程式员，以 public 方式继承一个 base，并以「保存 base class 的语意」的方式改写大部份虚拟函式。然而在此情况下，某些时候以 Derived object 做为一个 Base object，其行为并不完全是 Base 的 client 所期望。Robert Martin 常引用一个例子，从 Rectangle class 身上衍生下来 Square class。正方形是矩形，这在数学上或许为真，但在 classes 的世界中不一定为真。例如，假设 Rectangle class 有一个虚拟函式 SetWidth(int)。Square 设定宽度时应该很自然地也设定高度，使形状永远保持正方。但是程式某处有一些码，它们以多型方式（polymorphically）运用 Rectangle objects，它们并不预期宽度的改变也会影响高度，毕竟那对矩形而言不是真的。这是「public inheritance 违反 LSP」的一个好例子，因为 derived class 并未陈述与 base class 相同的语意。它违反了 public inheritance 的关键诫律：「不要求更多，也不承诺更少」。

当我看到人们做出这种"almost IS-A"，我通常会试著告诉他们，他们把自己带往麻烦之国。毕竟，某时某地某人会试著以多型方式来使用 derived objects，而偶尔会获得非预期的结果，不是吗？『但它没有问题』，有人这样回答，『它只是有一点点不相容罢了，没有人会以危险的方式这么使用 Base-family objects』。唔，所谓「一点点不相容」真是含蓄呀，现在，我没有理由怀疑这位程式员的正确性，对吗？也就是说此後这个系统之中再没有程式码可以冲击这个危险的差异了。然而我有足够的理由相信，某时某地将会有位维护工程师，当他打算做一点表面上无害的改变时，被这个问题缠住，并花费数个小时分析为什么这个 class 有如此差劲的设计，然後再花更多天的时间去修正它。

不要被诱惑，只要说 no 就是了。如果其行为不像 Base，它就不是一个（NOT-A）Base，那就不要以「使它看起来像个 Base」的方式来衍生它。

设计准则：总是确定 public inheritance 用以模塑 IS-A 和 WORKS-LIKE-A 的关系，并符合 Liskov Substitution Principle。所有被改写的 member functions 都必须「不要求更多，也不承诺更少」。

设计准则：绝对不要为了重复使用 base class 中的码而使用 public inheritance。Public inheritance 的目的是为了被既有的码以多型方式（polymorphically）重复运用 base objects。

结论

明智地运用 inheritance。如果你能够单独以 containment/delegation 表现出某个 class 相互关系，你应该那么做。如果你需要 inheritance 但不想模塑出 IS-A 的关系，请使用 nonpublic inheritance。如果你不需要多重继承（multiple inheritance）的威力，请使用单一继承（single inheritance）。一般而言，大而深的继承体系特别难以理解，因此也就难以维护。Inheritance 是一个设计期的决定，必须舍去许多执行期的弹性。

某些人以为，除非使用继承机制，否则不算物件导向。那不是真的。尽量使用可有效运作的解答中最简单的一个，那么你或许还有可能多享受几年稳定而容易维护的码。

条款 25：GotW#13 面向对象程式设计（Object-Oriented Programming）

难度：4

C++ 是一个物件导向语言吗？也是，也不是，和一般印象有点不同。为了改变步调与配速（并且在漫漫艰苦的程式码中暂时做个休息），下面是随笔式的提问。

C++ 是一个威力强大的语言，提供许多高级的物件导向架构，包括封装(encapsulation)、异常处理(exception handling)、继承（inheritance）、范本（templates）、多型（polymorphism）、强烈型别检验（strong typing），以及一个完全的模组系统（module system）。

试讨论之。

解答

这个条款的目的是为了诱发你思考主要的（即使是被遗漏的）C++ 特性。并提供有益的实务。更特别的是，我希望你对此条款的思考，得以产出一些观念和例子，用以说明三件主要事情。

1. 不是每个人都同意 OO 的意义。到底什么是物件导向呢？即使在今天，如果你问 10 个人，大概会获得 15 个不同的回答（不比拿法律问题问 10 个律师的情况好多少）。

几乎每个人都同意继承和多型是 OO 概念。大部份人还会加上封装。还有一些人可能加上异常处理，但大概没有人认为 templates 也是。对於什么是 OO 特性而什么不是，人言言殊，每一个观点都有热烈的拥护者。

2. C++ 是一个支援多种思维模式（multiparadigm）的语言。C++ 并不只是一个 OO 语言，它支援许多 OO 特性，但并不强迫程式员使用它们。你可以写完全没有 OO 精神的 C++ 程式，许多人正是如此。

C++ 标准化的努力过程中最重要的贡献就是让 C++ 更强烈地支援更具威力的抽象性（abstraction）以降低软体复杂度（Martin95）[注 12]。C++ 不只是一个物件导向语言。它支援数种编程风格，包括物件导向编程（object-oriented programming）和泛型编程（generic programming）。这些风格有其重要性，因为它们都提供了弹性的方法，可以帮助你透过抽象化来组织你的程式码。物件导向作法让我们将一个物件的状态（states）和处理那些状态的函式包装在一起。封装和继承让我们管理独立性并使重复可用性更清楚更容易。泛型（generic）方法是比较晚近的风格，让我们得以写出一些 functions 和 classes，操作其他涉及「未明定的、无任何关联的、未知的型别」的 functions 和 objects。泛型技术提供独特的方法，降低程式内的耦合性和依存性。现今其他某些语言也支援泛型（genericity），但还没有一个能够像 C++ 有如此强烈的支援。的确，新式泛型编程是因为 C++ 独一无二的 templates 特性才得以实现。

今天，C++ 提供了许多威力强大的方法来表现抽象性，而其所导致的弹性是 C++标准化的最重要成果。

（注 12： Martin95 这本书内有一份卓越的讨论，探讨为什么物件导向程式设计的最重要利益之一是，透过对程式码独立程度的管理，让我们降低软体的复杂度。）
3. 没有任何一个语言能够集所有功能和优点於一身。今天我使用 C++ 做为主要程式语言，明天我可能会使用更好的工具。是的，C++ 并没有模组系统（module system），也缺乏如垃圾收集（garbage collection）及某些重要性质；它有 static typing，但不一定称得上 strong typing。是的，每一个语言都有优点和缺点，你应该针对你的工作选择正确的工具，避免成为一个眼光浅薄的语言狂热份子。

7. 编译级防火墙及 Pimpl Idiom（Compiler Firewalls and the Pimpl Idiom）

条款 26～28：GotW#7 编译期的依赖性（Minimizing Compile-time Dependencies）

难度：7 / 10

（大多数程序员使用#include 包含的头文件都比实际需要的多。你也是这样的吗？想知道的话，请看本条款。）

[问题]

[注意：这个问题比想象的还要难！下面程序中的注释都是非常有用的。]

大多数程序员使用#include 包含的头文件都比实际需要的要多。这会严重的影响并延长程序的建立时间（build time），特别是当一个被频繁使用的头文件中包含了太多其它的头文件的时候，问题越发严重。

首先，在下面的头文件当中，有哪些#include 语句可以在不对程序产生副作用的情况下被直接去掉？其次，还有哪些#include 语句可以在对程序进行适当的修改之后被去掉？程序将如何修改？（你不能改变 X 类和 Y 类的公共接口；也就是说，你对这个头文件所作的任何修改都不能影响调用它的代码）。

```
// gotw007.h (implementation file is gotw007.cpp)
//
#include "a.h"  // class A
#include "b.h"  // class B
#include "c.h"  // class C
#include "d.h"  // class D
           // (注意: 只有 A 和 C 有虚拟函数（virtual functions）)
#include <iostream>
#include <ostream>
#include <sstream>
#include <list>
#include <string>
class X : public A {
public:
         X        ( const C& );
    D    Function1( int, char* );
    D    Function1( int, C );
    B&   Function2( B );
    void Function3( std::wostringstream& );
    std::ostream& print( std::ostream& ) const;
private:
    std::string  name_;
    std::list<C> clist_;
    D            d_;
};
std::ostream& operator<<( std::ostream& os, const X& x )
    { return x.print(os); }
class Y : private B {
public:
    C  Function4( A );
private:
    std::list<std::wostringstream*> alist_;
};
```

[解答]

首先，我们考虑那些可以被直接去掉的#include 语句（或者说是头文件）。为了便于查看，我们再把原始的代码列在下面：

```
// gotw007.h (其实现文件为 gotw007.cpp)
//
#include "a.h"  // class A
#include "b.h"  // class B
#include "c.h"  // class C
#include "d.h"  // class D
                // (注意: 只有 A 和 C 有虚拟函数（virtual functions）)
#include <iostream>
#include <ostream>
#include <sstream>
#include <list>
#include <string>
class X : public A {
public:
         X        ( const C& );
    D    Function1( int, char* );
    D    Function1( int, C );
    B&   Function2( B );
    void Function3( std::wostringstream& );
    std::ostream& print( std::ostream& ) const;
private:
    std::string  name_;
    std::list<C> clist_;
    D            d_;
};
std::ostream& operator<<( std::ostream& os, const X& x )
    { return x.print(os); }
class Y : private B {
public:
    C  Function4( A );
private:
    std::list<std::wostringstream*> alist_;
};
```

1. 我们可以直接去掉的头文件有：

   - iostream，因为程序里尽管用到了流，但并没有用到 iostream 里特定的东西。
   - ostream 和 sstream，因为程序中的参数和返回类型被前置声明（forward-declared）是可以的，所以其实只需要 iosfwd 就够了(要注意,并没有与 iosfwd 相对应的诸如 stringfwd 或者 listfwd 之类的标准头文件；iosfwd 是考虑到向下兼容性问题的产物，它使得以前那些不支持模板的流子系统的代码仍然可用而不需要修改或者重写。)

   我们不能直接去掉的头文件有：

   - a.h，因为 A 是 X 的基类。
   - b.h，因为 B 是 Y 的基类。
   - c.h，因为现有的许多编译器需要 list<C>能够看见对 C 的定义（这些编译器应该在未来的版本中修正这一点。）
   - d.h，list 和 string，因为 X 需要知道 D 和 string 的大小，X 和 Y 都需要知道 list 的大小。

   其次，我们再来考查那些可以通过隐藏 X 和 Y 的实现细节来被去掉的#include 语句（或者说是头文件）：

2. 我们可以通过让 X 和 Y 使用 pimpl_的方法来去掉 d.h，list 和 string（也就是说，其私有部分被一个指针代替，这个指针指向类型被前置声明（forward-declared）的实体对象），因为这时，X 和 Y 都不再需要知道 list、D 或者 string 的大小。这也使我们可以干掉 c.h，因为这时在 X::clist 中的 C 对象只作为参数或者返回值出现。

重要的事项：即使 ostream 没有被定义，内联的 operator<<也可能仍然保持其内联性并使用其 ostream 参数！这是因为你只在调用成员函数的时候才真正需要相应的定义；当你想接收一个对象，并只将其当成一个在其它

函数调用时的参数而不做任何其它额外的事情时，你并不需要该函数的定义。

最后，我们来看一下那些可以通过其它微小的修改而去掉的头文件：

3. 我们注意到 B 是 Y 的 private 基类，而且 B 没有虚拟函数（virtual function），因此 b.h 也是有可能被去掉的。有一个（也只有一个）主要的原因使我们在派生类的时候使用 private 继承，那就是想要重载（override）虚拟函数（virtual function）。如此看来，在这里与其让 Y 继承自 B，还不如让 Y 拥有一个类型为 B 的成员。要去掉 b.h，我们应该让 Y 的这个类型为 B 的成员存在于 Y 中隐藏的 pimpl_部分。

[学习指导]：请使用 pimpl_把代码的调用者与代码的实现细节隔离开来。

摘录自 GotW 的编码标准：

- **封装（encapsulation）和隔离（insulation）：**
- **在声明一个类的时候，应避免暴露出其私有成员：**
- **应该使用一个形如“struct Xxxxlmpl* pimpl_”的不透明的指针来存储私有成员（包括状态变量和成员函数），例如：**

```
class Map {private: struct Maplmpl* pimpl_;};
```

（Lakos96: 398-405; Meyers92: 111-116; Murray93: 72-74）

4. 基于以下几个原因，我们目前还不能够对 a.h 动手脚：A 被用作 public 基类；A 含有虚拟函数（virtual function），因而其 IS-A 关系可能会被代码的调用者所使用。然而我们注意到，X 和 Y 两个类之间没有任何关系，因此我们至少可以把 X 和 Y 的定义分别放到两个不同的头文件中间去（为了不影响现有的代码，我们还应该把现有的头文件作为一个存根（stub），让其用#include 包含 x.h 和 y.h）。如此以来，我们至少可以让 y.h 不用#include 包含 a.h，因为现在它只把 A 用作函数参数的类型，不需要 A 的定义。

综上所述，我们现在可以得到一个清爽的头文件了：

```
//-------------------------------------------------------------
// 新文件 x.h: 只包含两个#include!
//
#include "a.h"  // class A
#include <iosfwd>
class C;
class D;
class X : public A {
public:
        X       ( const C& );
    D   Function1( int, char* );
    D   Function1( int, C );
    B&  Function2( B );
    void Function3( std::wostringstream& );
    std::ostream& print( std::ostream& ) const;
private:
    class XImpl* pimpl_;
};
inline std::ostream& operator<<( std::ostream& os, const X& x )
    { return x.print(os); }
    // 注意: 这里不需要 ostream 的定义!
//-------------------------------------------------------------
// 新文件 y.h: 没有#include!
//
class A;
class C;
class Y {
public:
    C  Function4( A );
private:
```

```
    class YImpl* pimpl_;
};
//-----------------------------------------------------------------------
// gotw007.h 作为存根包含两个#include，又通过 x.h 附带了另外两个#include)
//
#include "x.h"
#include "y.h"
//-----------------------------------------------------------------------
// gotw007.cpp 中的新结构... 注意：impl 对象应该在 X 和 Y 的构造函数中
// 用 new 来创建，并在 X 和 Y 的析构函数中用 delete 来清除。
// X 和 Y 的成员函数要通过 pimpl_ 指针来访问数据
//
struct XImpl    // 是的, 我们可以用"struct" ，虽然前置声明的时候
{               // 我们用了"class"
    std::string  name_;
    std::list<C> clist_;
    D            d_;
}
struct YImpl
{
    std::list<std::wostringstream*> alist_;
    B b_;
}
```

最后说几句：到现在，X 的使用者只需要用#include 包含 a.h 和 iosfwd 就可以了。而 Y 的使用者也只需要包含 a.h 和 iosfwd，即使后来为了更新代码而需要包含 y.h 并去掉 gotw007.h，也照样是一行#include 都不用多加。与原来的程序相比，这是多么大的改进呀！

条款 29：GotW#24 编译级防火墙（Compilation Firewalls）

难度：6 / 10

（使用 pimpl 惯用法可以大大降低代码之间的相互依赖性，还可以减少程序的建立时间。但问题是，应该把那些东西放入 pimpl 对象里呢？如何才能安全的使用它呢？）

[问题]

在 C++中，如果类定义中的任何部分被改变了（即使是私有成员），那么这个类所有的使用者代妈都必须重新编译。为了降低这种依赖性，使用的一种常见的技术就是利用一个不透明指针（opaque pointer）来隐藏一部分实现细节：

```
class X {
public:
  /* ... 公有成员 ... */
protected:
  /* ... 保护成员？... */
private:
  /* ... 私有成员？... */
  class XImpl* pimpl_;  // 指向一个被前置声明了的（forward-declared）类
                        // 之不透明指针
};
```

[提问]

1. 那些部分应该放入 Ximpl？有四种常见的原则，它们是：

- 将全部私有数据（但不是函数）放入 Ximpl；
- 将全部私有成员（译注：即包括函数）放入 Ximpl；
- 将全部私有成员和保护成员放入 Ximpl；
- 使 Ximpl 完全成为原来的 X，将 X 编写为一个完全由简单的前置函数（forwarding functions）（一个句柄/本体的变体）组成的公共接口。

它们各有什么优缺点？你如何从中选择合适的？

2. Ximpl 需要一个指向 X 对象的"反向指针（back pointer）"吗？

[解答]

首先看两个定义：

- 可见类（visible class）：客户代吗所见并操纵的类（在这里即是 X）。
- pimpl：可见类中隐藏在一个透明指针（也称为 pimpl_）下的类实现（在这里即是 XImpl）。

在 C++中，如果类定义中的任何部分被改变了（即使是私有成员），那么这个类所有的使用者代妈都必须重新编

译。为了降低这种依赖性，使用的一种常见的技术就是利用一个不透明指针（opaque pointer）来隐藏一部分实现细节：

这是"句柄/本体"惯用法（handle/body idiom）的一种变体。如 Coplien[注 1]所记载，这种方法主要用于在共享代码的情况下进行引用计数（reference counting）。

正如 Lakos[注 2]所指出的那样，"句柄/本体"惯用法（表现为我所称为的"pimpl 惯用法"之形式；这样的叫法缘自给其特意取的、易发音的"pimpl_"指针[注 3]）对于打破编译期的依赖性也是非常有用的。本条款的解答集中讨论这种用法，其中有些讨论总的来讲并不适从于"句柄/本体"惯用法。

使用这个惯用法的主要代价是性能：

1. 每一个构造操作都须分配内存。自己定制的分配器（custom allocator）或许可以缓解内存的额外消耗，但这还不是涉及到更多的工作。

2. 每一个隐藏成员都需要一个额外的间接层来予以对其访问。（如果被访问的隐藏成员本身又使用到了一个"反向指针（back pointer）"来调用可见类中的函数，那么就会有双重的间接性。）

(1) 那些部分应该放入 Ximpl？有四种常见的原则，它们是：

- 将全部私有数据（但不是函数）放入 Ximpl；

这是个不错的开端，因为现在我们得以对任何只用作数据成员的类进行前置声明（forward-declare）（而不是使用#include 语句来包含类的真正声明--这会使客户代码对其形成依赖）。当然，我们通常可以做得更好。

- 将全部私有成员（译注：即包括函数）放入 Ximpl；

这（几乎）是我平常的用法。不管怎么说，在 C++中，"客户代码不应该也并不关心这些部分"就意味着"私有（private）"，而私有的东西则最好藏起来（在一些拥有更"自由宽大"之法律的北欧国家里的情况除外）。

对此有两条警告，其中的第一个也就是我在上一段中加上"almost"的原因：

1. 即使虚拟函数是私有的，你也不能把虚拟成员函数隐藏在 pimpl 类中。如果想要虚拟函数覆写基类中的同名虚拟函数，那么该虚拟函数就必须出现在真正的派生类中。如果虚拟函数不是继承而来的，那么为了让之后层级的派生类能够覆写它，其还是必须出现在可见类中。

2. 如果 pimpl 中的函数要使用其它函数，其可能需要一个指向可见对象的"反向指针（back pointer）"--这又增加了一层间接性。这个反向指针通常被约定俗成的称为 self_。

- 将全部私有成员和保护成员放入 Ximpl；

如此更进一步的做法其实是错误的。保护成员（protected members）绝不应该被放进 pimpl，因为这样做等于就是对其弃之不用。无论如何，保护成员（protected members）正是为了让派生类看到并使用而存在的，因此如果派生类无法看到或使用它们的话，它们也就基本上全无用处了。

- 使 Ximpl 完全成为原来的 X，将 X 编写为一个完全由简单的前置函数（forwarding functions）（一个句柄/本体的变体）组成的公共接口。

这只在少数几个有限的情况下有用，其好处是可以避免使用反向指针，因为所有的服务全部都在 pimpl 类之中提供了。其主要的缺点是，这样做一般会使得可见类对于继承而言全无用处，无论是作为基类还是派生类。

(2) Ximpl 需要一个指向 X 对象的"反向指针（back pointer）"吗？

很不幸，回答通常为"是的"。无论如何，我们会把每一个对象分裂成两部分--只因为我们要隐藏其中一部分。

当可见类中的函数被调用的时候，经常需要使用隐藏部分（译注：即 pimpl 部分）中的一些函数和数据，以便完成是调用着的请求。这挺好，也很合理。然而可能不太明显的情况是：在 pimpl 中的函数也经常必须调用可见类中的函数--通常是因为需要调用的函数是公有成员或虚拟函数。

[注 1]：James O. Coplien. Advanced C++ Programming Styles and Idioms (Addison-Wesley, 1992).
[注 2]：J. Lakos. Large-Scale C++ Software Design (Addison-Wesley, 1996).
[注 3]：我以前经常将其写作 impl_。与其等价的 pimpl_写法其实是有我的朋友和同事 Jeff Sumner 提出的；Jeff Sumner 同我一样对用于指针变量的匈牙利式"p"前缀（译注：即用于命名变量的匈牙利表示法；在该表示法中，指针变量名以"p"开头，意即"pointer"）颇为倾心，另外其还对恐怖的双关语有着独到的敏感（译注：pimpl 发音与单词"pimple（意即痤疮、丘疹、脓疱、疙瘩、粉刺）"相同）。

条款 30：GotW #28“Fast Pimpl”技术（The "Fast Pimpl" Idiom）

难度：6 / 10

采用一些称为“低依赖度”或“效能”方面的捷径，在很多时候颇有诱惑力，但它不总是好主意。这儿有个很精彩的方法能在客观上同时并安全的实现二者。

问题

标准的 malloc()和 new()调用的开销都是很大的。在下面的代码中，程序员最初在 class Y 中设计了一个类型 X 的成员：

```
// file y.h
#include "x.h"
class Y {
    /*...*/
    X x_;
};
```

```
// file y.cpp
Y::Y() {}
```
这个 class Y 的申明需要 class X 的申明已经可见（从 x.h 中）。要避免这个条件，程序员首先试图这么写：
```
// file y.h
class X;
class Y {
    /*...*/
    X* px_;
};
// file y.cpp
#include "x.h"
Y::Y() : px_( new X ) {}
Y::~Y() { delete px_; px_ = 0; }
```
这很好地隐藏了 X，但它造成了：当 Y 被广泛使用时，动态内存分配上的开销降低了性能。最终，我们这无畏的程序员偶然间发现了“完美”解决方案：既不需要在 y.h 中包含 x.h，也不需要动态内存分配（甚至连一个前向申明都不需要!）：
```
// file y.h
class Y {
    /*...*/
    static const size_t sizeofx = /*some value*/;
    char x_[sizeofx];
};
// file y.cpp
#include "x.h"
Y::Y() {
    assert( sizeofx >= sizeof(X) );
    new (&x_[0]) X;
}
Y::~Y() {
    (reinterpret_cast<X*>(&x_[0]))->~X();
}
```
讨论

解答

简短的答案

不要这样做。底线：C++不直接支持不定类型（opaque types），这是依赖于这一局限上的脆弱尝试（有些人甚至称此为“hack”[注 1]）。

程序员期望的肯定是其它一些东西，它被称为“Fast Pimpl”方法，我将在“为什么尝试 3 是糟糕的”一节中介绍。

为什么尝试 3 是糟糕的

首先，让我们稍许考虑一下为什么上面的尝试 3 是糟糕的，从以下方面：

1. 对齐。不象从::operator new()得到的动态内存，这个 x_的字符 buffer 并不确保满足类型 X 的对齐要求。试图让 x_工作得更可靠，程序员需要使用一个“max_align”联合，它的实现如下：
```
union max_align {
  short       dummy0;
  long        dummy1;
  double      dummy2;
  long double dummy3;
  void*       dummy4;
  /*...and pointers to functions, pointers to
       member functions, pointers to member data,
       pointers to classes, eye of newt, ...*/
};
```
它需要这样使用：
```
union {
  max_align m;
  char x_[sizeofx];
};
```
这不确保完全可移植，但在实际使用中，已经足够完美了，只在极少的系统（也可能没有）上，它不能如愿工作。

我知道有些老手认可甚至推荐这种技巧。凭良心，我还是称之为 hack，并强烈反对它。

2. 脆弱。Y 的作者必须极度小心处理 Y 的其它普通函数。例如，Y 绝不能使用默认赋值操作，必须禁止它或自己提供

一个版本。

写一个安全的 Y::operator=()不是很难，我把它留给读者作为习题。记得在这个操作和 Y::~Y()中考虑异常安全的需求。当你完成后，我想，你会同意这个方法制造的麻烦比它带来的好处远为严重。

3.维护代价。当 sizeof(X)增大到超过 sizeofx，程序员必须增大 sizeofx。这是一个没有引起注意的维护负担。选择一个较大的 sizeofx 值可以减轻这个负担，但换来了效能方面的损失（见#4）。

4.低效能的。只要 sizeofx 大于 sizeof(X)，空间被浪费了。这个损失可以降到最小，但又造成了维护负担（见#3）。

5.顽固不化。我把它放在最后，但不是最轻：简而言之，明显，程序员试图做些“与众不同”的事情。说实话，在我的经验中，“与众不同”和“hack”几乎是同意词。无论何时，只要见到这种“颠覆”行为——如本例的在字符数组中分配对象，或 GotW#23 中谈到的用显式的析构加 placement new 来实现赋值——，你都一定要说“No”。

我就是这个意思。好好反思一下。

更好的解决方法：“Fast Pimpl”

隐藏 X 的动机是避免 Y 的用户必须要知道（于是也就依赖）X。C++社区中，为了消除这种实现依赖，通常使用“pimpl”方法（注 2），也就是我们这个无畏的程序员最初所尝试的方法。

唯一的问题是，“pimpl”方法由于为 X 的对象在自由空间分配内存而导致性能下降。通常，对特殊类的内存分配性能问题，采用为它提供一个 operator new 的重载版本的方法，因为固定尺寸的内存分配器可以比通用内存分配器性能高得多。

不幸的是，这也意味着 Y 的作者必须也是 X 的作者。通常，这不成立。真正的解决方法是使用一个高效的 Pimpl，也就是，有自己的类属 operator new 的 pimpl 类。

```
// file y.h
class YImpl;
class Y {
  /*...*/
  YImpl* pimpl_;
};
// file y.cpp
#include "x.h"
struct YImpl {  // yes, 'struct' is allowed :-)
  /*...private stuff here...*/
  void* operator new( size_t )   { /*...*/ }
  void  operator delete( void* ) { /*...*/ }
};
Y::Y() : pimpl_( new YImpl ) {}
Y::~Y() { delete pimpl_; pimpl_ = 0; }
```

“啊!”你叫道，“我们找到了圣杯——Fast Pimpl!”。好，是的，但先等一下，先想一下它是如何工作的，并且它的代价是什么。

你所收藏的 C++宝典中肯定展示了如何实现一个高效的固定尺寸内存分配/释放函数，所以我就不多罗嗦了。我要讨论的是“可用性”：一个技巧，将它们放入一个通用固定尺寸内存分配器模板中的技巧，如下：

```
template<size_t S>
class FixedAllocator {
public:
  void* Allocate( /*requested size is always S*/ );
  void  Deallocate( void* );
private:
  /*...implemented using statics?...*/
};
```

因为它的私有细节很可能使用 static 来实现，于是，问题是 Deallocate()是否被一个 static 对象的析构函数调用（Because the private details are likely to use statics, however, there could be problems if Deallocate() is ever called from a static object's dtor)。也许更安全的方法是使用 single 模式，为每个被请求的尺寸使用一个独立链表，并用一个唯一对象来管理所有这些链表（或者，作为效能上的折中，为每个尺寸“桶”使用一个链表；如，一个链表管理大小在 0～8 的内存块，另一个链表表管理大小在 9～16 的内存块，等等。）：

```
class FixedAllocator {
public:
  static FixedAllocator* Instance();
  void* Allocate( size_t );
  void  Deallocate( void* );
private:
    /*...singleton implementation, typically
         with easier-to-manage statics than
         the templated alternative above...*/
};
```

让我们用一个辅助基类来封装这些调用：

```
struct FastPimpl {
```

```
  void* operator new( size_t s ) {
    return FixedAllocator::Instance()->Allocate(s);
  }
  void operator delete( void* p ) {
    FixedAllocator::Instance()->Deallocate(p);
  }
};
```

现在，你可以很容易地写出你任意的 Fast Pimpl：

```
//  Want this one to be a Fast Pimpl?
//  Easy, then just inherit...
struct YImpl : FastPimpl {
  /*...private stuff here...*/
};
```

但，小心！

这虽然很好，但也别乱用 Fast Pimpl。你得到了最佳的内存分配速度，但被望了代价：维护这些独立的链表将导致空间效能的下降，因为这比通常情况现成更多的内存碎片（Managing separate free lists for objects of specific sizes usually means incurring a space efficiency penalty because any free space is fragmented (more than usual) across several lists)。

和其它性能优化方法一样，只有在使用了 profiler 剖析性能并证明需要性能优化后，才使用这个方法。

（注 1. 我不是其中之一。 :-))

（注 2. 参见 GotW #24.）

8. 名称搜索、命名空间、接口原则（Name Lookup, Namespaces, and the Interface Principle）

条款 31：GotW#30 名称搜索（Name Lookup）

难度：9.5 / 10

当你调用一个函数时，到底调的是哪一个？其答案取决于“名称搜索”，但你肯定会发现其细节非常令人吃惊。

问题

在下面的代码中，调用的是哪个函数？为什么？分析一下影响。

```
namespace A {
  struct X;
  struct Y;
  void f( int );
  void g( X );
}
namespace B {
    void f( int i ) {
        f( i );   // which f()?
    }
    void g( A::X x ) {
        g( x );   // which g()?
    }
    void h( A::Y y ) {
        h( y );   // which h()?
    }
}
```

解答

在下面的代码中，调用的是哪个函数？为什么？

其中两个比较明确，但第三个需要精通 C++的名称搜索规则－－尤其是 Koenig lookup。

```
namespace A {
  struct X;
  struct Y;
  void f( int );
  void g( X );
}
namespace B {
    void f( int i ) {
        f( i );   // which f()?
```

```
    }
```

这个 f()调用的是它自己，并且是无穷递归。

在 namespace A 中也有一个 f(int)的函数。如果 B 写了“using namespace A;”或“using A::f;”，那么 A::f(int)将是寻找 f(int)过程中的候选函数之一，那个“f(i)”调用将在 A::f(int)和 B::f(int)间造成二义性。因为 B 没有将 A::f(int)带入它的空间范围，于是只有 B::f(int)被考虑，所以调用没有歧义。

```
    void g( A::X x ) {
        g( x );   // which g()?
    }
```

这个调用在 A::g(X)和 B::g(X)间有二义性。程序员必须用命名空间的名字明确限定调用哪个 g()。

你也许开始会奇怪为什么需要这么做。毕竟，和 f()一样，B 没有用“using”指令将 A::g(X)带入它的空间范围，所以，你可能认为只有 B::g(X)可选。没说错吧？好吧，这是个事实，在名称搜索时的一条补充规则：

Koenig Lookup （简化版）

如果你给函数提供一个 class 类型的实参（此处是 A::X 类型的 x），那么在名称搜索时，编译器将认为包含实参类型的命名空间中的同名函数为可选函数。

下面这个更复杂些，但实质是一样的。这是取自 C++标准的例子：

```
namespace NS {
    class T { };
    void f(T);
}
NS::T parm;
int main() {
    f(parm);    // OK, calls NS::f
}
```

我于此处不深究 Koenig lookup 的必要性（要延伸你的思维的话，将“NS”用“std”代替，将“T”用“string”代替，而“f”用“operator<<”代替，然后再考虑流程）。在最后的“延伸读物”中寻找 Koenig lookup 的更多知识，以及它在名称隔离和分析类间依赖上的影响。必须承认 Koenig lookup 实际上是个好东西，你也必须了解它是如何工作的，因为它将在某个时候影响你写下的代码。

```
    void h( A::Y y ) {
        h( y );   // which h()?
    }
}
```

没有其它的 h(A::Y)函数，所以这儿的 f()调用它自己，也是无穷递归。

虽然 B::h()的原型是使用了 namespace A 中的一个类型，但这不影响名称搜索的结果，因为 namespace A 中没有符合 h(A::Y)名称的函数。

分析影响

要之，namespace B 中的代码的含义被完全独立的 namespace A 中申明的函数影响了，虽然 B 在简单地提及了一个在 A 中找到的类型以外，没干任何事情，甚至连“using”都没有！

这意味着，命名空间之间不象人们最初想象得那样毫无依赖关系。别急着谴责命名空间；命名空间之间的互不依赖还是很完美的，它们之间只用一个 T 类型建立联系。本期的 GotW 只是想指出这样一个特例，此时，命名空间不是密不透风的...并且，实际上，在这种情况下，命名空间不应该是密不透风的，“延伸读物”将展示这一点。

延伸读物

命名空间与 Koenig lookup 之间的影响远比我所展示的深远。想得知为什么 Koenig lookup 不是命名空间上的漏洞，以及它怎样影响我们分析 class 之间的依赖关系，参阅我在 March 1998 于 C++ Report 上发表的文章《What's In a Class - The Interface Principle》。

条款 32～34：接口原则（the Interface Principle）

类里面是什么？－接口原则

This article appeared in C++ Report, 10(3), March 1998.

我开始于一个简单而令人困惑的问题：

- 类里面是什么？也就是，什么组成了类及其接口？

更深层的问题是：

- 它怎样和 C 风格的面向对象编程联系起来？
- 它怎样和 C++的 Koenig lookup 联系起来？和称为 Myers Example 的例子又有什么联系？（我都会讨论。）
- 它怎样影响我们分析 class 间的依赖关系以及设计对象模型？

那么，“class 里面是什么？”

首先，回顾一下一个 class 的传统定义：

Class（定义）

一个 class 描述了一组数据及操作这些数据的函数。

程序员们常常无意中曲解了这个定义，改为：“噢，一个 class，也就是出现在其定义体中的东西－－成员数据

和成员函数。”于是事情变味了，因为它限定“函数”为只是“成员函数”。看这个：

```
//*** Example 1 (a)
class X { /*...*/ };
/*...*/
void f( const X& );
```

问题是：f 是 X 的一部分吗？一些人会立即答“No”，因为 f 是一个非成员函数（或“自由函数”）。另外一些人则认识到关键性的东西：如果 Example 1 (a)的代码出现在一个头文件中，也就是：

```
//*** Example 1 (b)
class X { /*...*/
public:
  void f() const;
};
```

再考虑一下。不考虑访问权限〖注 1〗，f 还是相同的，接受一个指向 X 的指针或引用。只不过现在这个参数是隐含的。于是，如果 Example 1 (a)全部出现在同一头文件中，我们将发现，虽然 f 不是 X 的成员函数，但强烈地依赖于 X。我将在下一节展示这个关联关系的真正含义。

换种方式，如果 X 和 f 不出现在同一头文件中，那么 f 只不过是个早已存在的用户函数，而不是 X 的一部分（即使 f 需要一个 X 类型的实参）。我们只是例行公事般写了一个函数，其形参类型来自于库函数头文件，很明确，我们自己的函数不是库函数中的类的一部分。

The Interface Principle

接口原则（Interface Principle）

依靠那个例子，我将提出接口原则：

接口原则

对于一个类 X，所有的函数，包括自由函数，只要同时满足

(a) “提到”X，并且

(b) 与 X“同期提供”

那么就是 X 的逻辑组成部分，因为它们形成了 X 的接口。

根据定义，所有成员函数都是 X 的“组成部分”：

(a) 每个成员函数都必须“提到”X（非 static 的成员函数有一个隐含参数，类型是 X*（WQ：C++标准制定后，精确地说，应该是 X* const）或 const X*（WQ：const X* const）；static 的成员函数也是在 X 的作用范围内的）；并且

(b) 每个成员函数都与 X“同期提供”（在 X 的定义体中）。

用接口原则来分析 Example 1 (a)，其结论和我们最初的看法相同：很明确，f“提到”X。如果 f 也与 X“同期提供”（例如，它们存在于相同的头文件和/或命名空间中〖注 2〗），根据接口原则，f 是 X 的逻辑组成部分，因为它是 X 的接口的一部分。

接口原则在判断什么是一个 class 的组成部分时，是个有用的试金石。把自由函数判定为一个 class 的组成部分，这违反直觉吗？那么，给这个例子再加些砝码，将 f 改得更常见些：

```
//*** Example 1 (c)
class X { /*...*/ };
/*...*/
ostream& operator<<( ostream&, const X& );
```

现在，接口原则的基本理念就很清楚了，因为我们理解这个特别的自由函数是怎么回事：如果 operator<<与 X“同期提供”（例如，它们存在于相同的头文件和/或命名空间中），那么，operator<<是 X 的逻辑组成部分，因为它是 X 的接口的一部分。

据此，让我们回到 class 的传统定义上：

Class（定义）

一个 class 描述了一组数据及操作这些数据的函数。

这个定义非常正确，因为它没有说到“函数”是否为“成员”。

这个接口原则是 OO 的原则呢，还是只是 C++特有的原则？

我用了 C++的术语如“namespace”来描述“与...同期提供”的含义的，所以，接口原则是 C++特有的？还是它是 OO 的通行原则，并适用于其它语言？

考虑一下这个很类似的例子，从另一个语言（实际上，是个非 OO 的语言）：C。

```
/*** Example 2 (a) ***/
struct _iobuf { /*...data goes here...*/ };
typedef struct _iobuf FILE;
FILE* fopen ( const char* filename,
              const char* mode );
int   fclose( FILE* stream );
int   fseek ( FILE* stream,
              long  offset,
              int   origin );
long  ftell ( FILE* stream );
     /* etc. */
```

这是在没有 class 的语言中实现 OO 代码的典型“处理技巧”：提供一个结构来包容对象的数据，提供函数——肯定是非成员的——接受或返回指向这个结构的指针。这些自由函数构造（fopen），析构（fclose）和操作（fseek、ftell 等等……）这些数据。

这个技巧是有缺点的（例如，它依赖于用户能忍住不直接胡乱操作数据），但它“确实”是 OO 的代码——总之，一个 class 是一组数据及操作这些数据的函数。虽然这些函数都是非成员的，但它们仍然是 FILE 的接口的一部分。

现在，考虑一下怎样将 Example 2 (a)用一个有 class 的语言重写：

```
//*** Example 2 (b)
class FILE {
public:
  FILE( const char* filename,
        const char* mode );
 ~FILE();
  int  fseek( long offset, int origin );
  long ftell();
       /* etc. */
private:
  /*...data goes here...*/
};
```

那个 FILE*的参数变成为隐含参数。现在就明确了，fseek 是 FILE 的一部分，和 Example 2 (a)中它还不是成员函数时一样。我们甚至可以将函数实现为一部分是成员函数，一部分是非成员函数：

```
//*** Example 2 (c)
class FILE {
public:
  FILE( const char* filename,
        const char* mode );
 ~FILE();
  long ftell();
       /* etc. */
private:
  /*...data goes here...*/
};
int fseek( FILE* stream,
           long  offset,
           int   origin );
```

很清楚，是不是成员函数并不是问题的关键。只要它们“提及”了 FILE 并且与 FILE“同期提供”，它们就是 FILE 的组成部分。在 Example 2 (a)中，所有的函数都是非成员函数，因为在 C 语言中，它们只能如此。即使在 C++中，一个 class 的接口函数中，有些必须（或应该）是非成员函数：operator<<不能是成员函数，因为它要求一个流对象作为左操作参数，operator+不该是成员函数以允许左操作参数发生（自动）类型转换。

介绍 Koenig Lookup

接口原则还有更深远的意义，如果你认识到它和 Koenig Lookup 干了相同的事的话〖注 4〗。此处，我将用两个例子来解说和定义 Koenig Lookup。下一节，我将用 Myers Example 来展示它为何与接口原则有直接联系。

这是我们需要 Koenig lookup 的原因，来自于 C++标准中的例子：

```
//*** Example 3 (a)
namespace NS {
  class T { };
  void f(T);
}
NS::T parm;
int main() {
  f(parm);    // OK: calls NS::f
}
```

很漂亮，不是吗？“明显”，程序员不需要明确地写为 NS::f(parm)，因为 f(parm)“明确”意味着 NS::f(parm)，对吧？但，对我们显而易见的东西对编译器来说并不总是显而易见，尤其是考虑到连个将名称 f 带入代码空间的“using”都没用。Koenig lookup 让编译器得以完成正确的事情。

它是这么工作的：所谓“名称搜索”就是当你写下一个“f(parm)”调用时，编译器必须要决策出你想调哪个叫 f 的函数。（由于重载和作用域的原因，可能会有几个叫 f 的函数。）Koenig lookup 是这么说的，如果你传给函数一个 class 类型的实参（此处是 parm，类型为 NS::T），为了查找这个函数名，编译器被要求不仅要搜索如局部作用域这样的常规空间，还要搜索包含实参类型的命名空间（此处是 NS）〖注 5〗。于是，Example 3 (a)中是这样的：传给 f 的参数类型为 T，T 定义于 namespace NS，编译器要考虑 namespace NS 中的 f——不要大惊小怪了。

不用明确限定 f 是件好事，因为这样我们就很容易限定函数名了：

```
//*** Example 3 (b)
#include <iostream>
```

```
#include <string>  // this header
    //   declares the free function
    //   std::operator<< for strings
int main() {
  std::string hello = "Hello, world";
  std::cout << hello; // OK: calls
}                     //   std::operator<<
```

这种情况下，没有 Koenig lookup 的话，编译器将无法找到 operator<<，因为我们所期望的 operator<<是个自由函数，我们只知道它是 string 包的一部分。如果程序员被强迫为必须限定这个函数名的话，会很不爽，因为最后一行就不能很自然地使用运算符了。取而代之的，我们必须写为“std::operator<<(std::cout, hello);”或“using namespace std;”。如果这种情况触痛了你的话，你就会明白为什么我们需要 Koenig lookup 了。

总结：如果你在同一命名空间中提供一个 class 和一个“提及”此 class 的自由函数〖注 6〗，编译器在两者间建立一个强关联〖注 7〗。再让我们回到接口原则上，考虑这个 Myers Example：

更多的 Koenig Lookup：Myers Example

考虑第一个（略为）简化的例子：

```
//*** Example 4 (a)
namespace NS { // typically from some
  class T { }; //   header T.h
}
void f( NS::T );
int main() {
  NS::T parm;
  f(parm);     // OK: calls global f
}
```

Namespace NS 提供了一个类型 T，而在它外面提供了一个全局函数 f，碰巧此函数接受一个 T 的参数。很好，天空是蔚蓝的，世界充满和平，一切都很美好。

时间在流逝。有一天，NS 的作者基于需要增加了一个函数：

```
//*** Example 4 (b)
namespace NS { // typically from some
  class T { }; //   header T.h
  void f( T ); // <-- new function
}
void f( NS::T );
int main() {
  NS::T parm;
  f(parm);     // ambiguous: NS::f
}              //   or global f?
```

在命名空间中增加一个函数的行为“破坏”了命名空间外面的代码，即使用户代码没有用“using”将 NS 中的名称带到它自己的作用域中！但等一下，事情是变好了——Nathan Myers〖注 8〗指出了在命名空间与 Koenig lookup 之间的有趣行为：

```
//*** The Myers Example: "Before"
namespace A {
  class X { };
}
namespace B {
  void f( A::X );
  void g( A::X parm ) {
    f(parm);   // OK: calls B::f
  }
}
```

很好，天很蓝……。一天，A 的作者基于需要增加了另外一个函数：

```
//*** The Myers Example: "After"
namespace A {
  class X { };
  void f( X ); // <-- new function
}
namespace B {
  void f( A::X );
  void g( A::X parm ) {
    f(parm);   // ambiguous: A::f or B::f?
  }
}
```

“啊？”你可能会问。“命名空间卖点就是防止名字冲突，不是吗？但，在一个命名空间中增加函数却看起来造成'破坏'了另一个完全隔离的命名空间中的代码。”是的，namespace B 中的代码被破坏了，只不过是因为它“提及”了来自于 namespace A 中的一个类型。B 中的代码没有在任何地方写出“using namespace; A”，也没有写出“using A::X;”。

这不是问题，B 没有被“破坏”。事实上，这是应该发生的正确行为〖注 9〗。如果在 X 所处的命名空间中有应该函数 f(X)，那么，根据接口原则，f 是 X 的接口的一部分。f 是一个自由函数根本不是关键；想确认它仍然是 X 的在逻辑组成部分，只要给它另外一个名字：

```
//*** Restating the Myers Example: "After"
namespace A {
  class X { };
  ostream& operator<<( ostream&, const X& );
}
namespace B {
  ostream& operator<<( ostream&, const A::X& );
  void g( A::X parm ) {
    cout << parm; // ambiguous:
  }               //   A::operator<< or
}                 //   B::operator<<?
```

如果用户代码提供了一个“提及”X 的函数，而它与 X 所处的命名空间提供的某个函数签名重合时，调用将有二义性。B 必须明确表明它想调用哪个函数，它自己的还是与 X“同期提供”的。这正是我们期望接口原则应该提供的东西：

接口原则

对于一个类 X，所有的函数，包括自由函数，只要同时满足

(a) “提到”X，并且

(b) 与 X“同期提供”

就是 X 的逻辑组成部分，因为它们组成了 X 的接口。

简而言之，接口原则与 Koenig lookup 的行为相同并不是意外。Koenig lookup 的行为正是建立在接口原则的基础上的。

（下面“关联有多强？”这节展示了为什么成员函数 class 之间仍然有比非成员函数更强的关联关系）

“组成部分”的关联有多强？

虽然接口原则说明成员和非成员函数都是 class 的逻辑“组成部分”，但它并没说成员和非成员是平等的。例如，成员函数自动得到 class 内部的全部访问权限，而非成员函数只有在它们被申明为友元时才能得到相同的权限。同样，在名称搜索（包括 Koenig lookup）中，C++语言特意表明成员函数与 class 之间有比非成员函数更强的关联关系：

```
//*** NOT the Myers Example
namespace A {
  class X { };
  void f( X );
}
class B {
// class, not namespace
 void f( A::X );
 void g( A::X parm ) {
 f(parm); // OK: B::f,
          // not ambiguous
 }
};
```

我们现在讨论的是 class B 而不是 namespace B，所以这没有二义：当编译器找到一个叫 f 的成员函数时，它不会自找麻烦地使用 Koenig lookup 来搜索自由函数。

所以，在两个主要问题上——访问权限规则和名称搜索规则——即使根据接口原则，当一个函数是一个 class 的组成部分时，成员函数与 class 之间有比非成员函数更强的关联关系。

一个 class 依赖于什么？

“class 里面是什么”并不只是一个哲理问题。它是一个根基性问题，因为，没有正确的答案的话，我们就不能恰当地分析 class 的依赖关系。

为了证明这一点，看一下这个似乎不相关的问题：怎么实现一个 class 的 operator<<？有两个主要方法，但都有取舍。我都进行分析，最后我们将发现我们又回到了接口原则上，它在正确分析取舍时提供了重要的指导原则。

第一种：

```
//*** Example 5 (a) -- nonvirtual streaming
class X {
  /*...ostream is never mentioned here...*/
};
ostream& operator<<( ostream& o, const X& x ) {
```

```
  /* code to output an X to a stream */
  return o;
}
```

这是第二个：

```
//*** Example 5 (b) -- virtual streaming
class X { /*...*/
public:
  virtual ostream& print( ostream& o ) {
    /* code to output an X to a stream */
    return o;
  }
};
ostream& operator<<( ostream& o, const X& x ) {
  return x.print();
}
```

假设两种情况下，class 和函数都出现在相同的头文件和/或命名空间中。你选择哪一个？取舍是什么？历来，C++的资深程序员用这种方式分析了这些选则：

- 选择(a)的好处是 X 有更低的依赖性。因为 X 没有成员函数“提及”了“流（ostream)”，X（看起来）不依赖于流。选择(a)同样也避免了一个额外的虚函数调用的开销。
- 选择(b)的好处是所有 X 的派生类都能正确 print，即使传给 operator<<的是一个 X&。

这个分析是有瑕疵的。运用接口原则，我们能发现为什么－－选择(a)的好处是假象，因为：

- 根据接口原则，只要 operator<<“提及”了 X（两种情况下都是如此）并且与 X“同期提供”（两种情况下都是如此），它就是 X 的逻辑组成部分。
- 两种情况下，operator<<都“提及”了流，所以 operator<<依赖流。
- 因为两种情况下 operator<<都是 X 的逻辑组成部分且都依赖于流，所以，两种情况下 X 都依赖流。

所以，我们素来认为的选择(a)的主要好处根本就不存在－－两种情况下 X 都依赖于流！如果（通常也如此）operator<<和 X 出现在相同的 X.h 中，两种情况下 X 的实现体和使用了 X 的实体的用户模块都依赖流，并至少需要流的前向申明以完成编译。

随着第一大好处的幻像的破灭，选择(a)就只剩下没有虚函数调用开销的好处了。不借助于接口原则的话，我们就没法如此容易地在这个很常见的例子上分析依赖性上的真象（以及事实上的取舍）。

底线：区分成员还是非成员没有太大的意义（尤其是在分析依赖性时），而这正是接口原则所要阐述的。

一些有趣（甚至是诧异）的结果

通常，如果 A 和 B 都是 clss，并且 f(A,B)是一个自由函数：

- 如果 A 与 f 同期提供，那么 f 是 A 的组成部分，并且 A 将依赖 B。
- 如果 B 与 f 同期提供，那么 f 是 B 的组成部分，并且 B 将依赖 A。
- 如果 A、B、f 都是同期提供的，那么 f 同时是 A 和 B 的组成部分，并且 A 与 B 是循环依赖。这具有根本性的意义－－如果一个库的作者提供了两个 class 及同时涉及二者的操作，那么这个操作恐怕被规定为必须同步使用。现在，接口原则对这个循环依赖给出了一个严格的证明。

最后，我们到了一个真的很有趣的状态。通常，如果 A 和 B 都是 class，并且 A::g(B)是 A 的一个成员函数：

- 因为 A::g(B)的存在，很明显，A 总是依赖 B。没有疑义。
- 如果 A 和 B 是同期提供的，那么 A::g(B)与 B 当然也是同期提供的。于是，因为 A::g(B)同时满足“提及”B 和与 B“同期提供”，根据接口原则（恐怕有些诧异）：A::g(B)是 B 的组成部分，而又因为 A::g(B)使用了一个（隐含的）A*参数，所以 B 依赖 A。因为 A 也依赖 B，这意味着 A 和 B 循环依赖。

首先，它看起来只是“将一个 class 的成员函数判定为也是另一个 class 的组成部分”的推论，但这只在 A 和 B 是同期提供时才成立。再想一下：如果 A 和 B 是同期提供的（也就是说，在同一头文件中），并且 A 在一个成员函数中提及了 B，“直觉”也告诉我们 A 和 B 恐怕是循环依赖的。它们之间肯定是强耦合的，它们同期提供和互相作用的事实意味着：(a)它们应该同步使用，(b)更改一个也会影响另一个。

问题是：在此以前，除了“直觉”很难用实物证明 A 与 B 间的循环依赖。现在，这个循环依赖可以用接口原则的来推导证明了。

注意：与 class 不同，namespace 不需要一次申明完毕，这个“同期提供”取决于 namespace 当前的可见部分：

```
//*** Example 6 (a)
//---file a.h---
namespace N { class B; } // forward decl
namespace N { class A; } // forward decl
class N::A { public: void g(B); };
//---file b.h---
namespace N { class B { /*...*/ }; }
```

A 的用户包含了 a.h，于是，A 和 B 是同期提供的并是循环依赖的。B 的用户包含了 b.h，于是 A 和 B 不是同期提供的。

总结

我希望你得到 3 个想法：

- 接口原则：对于 class X，所有的函数，包括自由函数，只要同时满足(a)“提及”X，(b)与 X“同期提供”，

那么它就是 X 的逻辑组成部分，因为它们是 X 的接口的一部分。

- 因此，成员和非成员函数都是一个 class 的逻辑组成部分。只不过成员函数比非成员函数有更强的关联关系。
- 在接口原则中，对“同期提供”的最有用的解释是“出现在相同的头文件和/或命名空间中”。如果函数与 class 出现在相同的头文件中，在依赖性分析时，它是此 class 的组成部分。如果函数与类出现在相同的命名空间中，在对象引用和名称搜索时，它是此 class 的组成部分。

注 1.即使最初 f 是一个友元，情况还是这样的。

注 2.我们在本文后面的篇幅中将详细讨论命名空间之间的关联关系，因为它指出了接口原则实际上和 Koenig lookup 的行为是一致的。

注 3.成员和非成员函数间的相似性，在另外一些可重载的操作符上表现得甚至更强烈。例如，当你写下“a + b”时，你可以调用 a.operator+(b)也可以是 operator+(a,b)，这取决于 a 和 b 的类型。

注 4.以 Andrew Koenig 命名的,因为最初由他写出了这个定义,他是 AT&T's C++ team 和 C++ standards committee 的长期会员。参见由 A. Koenig 和 B. Moo 写的《Ruminations on C++》(Addison-Wesley, 1997)。

注 5.还有其它一些细节，但本质就是这个。

注 6.通过传值、传引用、传指针，或其它方式。

注 7.要承认，其关联关系要比 class 与其成员间的关联关系要弱那么一点点。后文有“关联有多强”这么一节。

注 8.Nathan 也是 C++ standards committee 的长期会员，并且是标准中的 locale facility 的主要作者。

注 9.这个特别的例子出现在 November 1997 的 Morristown 会议上，它激起我思考成员关系和依赖关系。Myers Example 的意思很简单：命名空间不象人们通常想象的那样密不透风，但它们在隔离性上已足够完美并恰好满足它们本职工作。

9. 内存管理（Memory Management）

条款 35：GotW#9 内存管理（Memory Management）（一）

难度：3 / 10

（本条款介绍 C++中几个主要内存区域的基本知识。条款 10 将会继续本条款的话题,深入讨论一些内存管理的问题。）

[问题]

C++拥有几个不同的内存区域，用来存储对象或其它类型的值。每一个区域都有其各自的特点。

请叫出尽可能多的内存区域的名称，并分析每一个区域的性能特征，描述存储在其中的对象的生存周期。

举例：堆栈区（stack）存储自动变量（automatic variables），包括内建类型和类对象等。

[解答]

下面总结出了 C++程序主要的内存区域。注意，有些名称（比如 heap）可能与 C++标准中的叫法不一样。

- [常量数据（const data）区：]

常量数据区存储字符串等在编译期间就能确定的值。类对象不能存在于这个区域中。在程序的整个生存周期内，区域中的数据都是可用的。

区域内所有的数据都是只读的，任何企图修改本区域数据的行为都会造成无法预料的后果。之所以会如此，是因为在实际的实现当中，即使是最底层的内部存储格式也受制于所实现的特定的优化方案。例如，一种编译器完全可以把字符串存放在几个重叠的对象里面——只要实现者愿意的话。

- [栈（stack）区：]

栈区存储自动变量（automatic variables）。一般来说，栈区的分配操作要比动态存储区（比如堆（heap）或者自由存储区（free store））快得多，这是因为栈区的分配只涉及到一个指针的递增，而动态存储区的分配涉及到较为复杂的管理机制。栈区中，内存一旦被分配，对象就立即被构造好了；对象一旦被销毁，分配的内存也立即被收回（译注：这里作者用了“去配（deallocate）”一词，鄙人一律翻译为“回收”）。因此，在栈区中，程序员没有办法直接操纵那些已经被分配但还没有被初始化的栈空间（当然，那些通过使用显式（explicit）析构函数（destructor）和 new 运算符而故意这么做的情况不算在内）。

- [自由存储区（free store）：]

自由存储区（free store）是C++两个动态内存区域之一，使用new和delete来予以分配和释放（freed）。在自由存储区（free store）中，对象的生存周期可以比存放它的内存区的生存周期短；这也就是说，我们可以获得一片内存区而不用马上对其进行初始化；同时，在对象被销毁之后，也不用马上收回其占用的内存区。在对象被销毁而其占用的内存区还未被收回的这段时间内，我们可以通过 void*型的指针访问这片区域，但是其原始对象的非静态成员以及成员函数（即使我们知道了它们的地址）都不能被访问或者操纵。

- [堆（heap）区：]

堆（heap）区是另一个动态存储区域，使用malloc、free以及一些相关变量来进行分配和回收。要注意，虽然在特定的编译器里缺省的全局运算符new和delete也许会按照malloc和free的方式来被实现，但是堆（heap）与自由存储区（free store）是不同的——在某一个区域内被分配的内存不可能在另一个区域内被安全的回收。堆（heap）中被分配的内存一般用于存放在使用new的构造过程中和显式（explicit）的析构过程中涉及到的类对象。堆中对象的生存周期与自由存储区（free store）中的类似。

- [全局/静态区（Global/Static）：]

全局的或静态的变量和对象所占用的内存区域在程序启动（startup）的时候才被分配，而且可能直到程序开始执行的时候才被初始化。比如，函数中的静态变量就是在程序第一次执行到定义该变量的代码时才被初始化的。对那些跨越了翻译单元（translation unit）的全局变量进行初始化操作的顺序是没有被明确定义的，因而需要特别注意管理全局对象（包括静态类对象）之间的依赖关系。最后，和前面讲的一样，全局/静态区（Global/Static）中没有被初始化的对象存储区域可以通过 void*来被访问和操纵，但是只要是在对象真正的生存周期之外，非静态成员和成员函数是无法被使用或者引用的。

[关于“堆（heap）vs.自由存储区（free store）”]：本条款中我们将堆（heap）和自由存储区（free store）区分开来，是因为在C++标准草案中，关于这两种区域是否有联系的问题一直很谨慎的没有予以详细说明。比如当内存在通过delete运算符进行回收时，18.4.1.1中说道：

It is unspecified under what conditions part or all of such reclaimed storage is allocated by a subsequent call to operator new or any of calloc, malloc, or realloc, declared in <cstdlib>.
[关于在何种情况下，这种内存区域的部分或全部才会通过后续的对 new（或者是在<cstdlib>里声明的 calloc，malloc，realloc中的任何一个）的调用来被分配的问题，在此不作详细规定，不予详述。]

同样，在一个特定的实现中，到底new和delete是按照malloc和free来实现的，或者反过来malloc和free是按照new和delete来实现的，这也没有定论。简单地说吧，这两种内存区域运作方式不一样，访问方式也不一样——所以嘛，当然应该被当成不一样的两个东西来使用了！

条款36：GotW#10 内存管理（Memory Management）（二）

难度：6 / 10

（你在考虑对某个类实现你自己特定的内存管理方案吗？甚或是想干脆替换掉C++的全局操作符new和delete？先试试下面这个问题再说吧！）

[问题]

下面的代码摘自某个程序，这个程序有一些类使用它们自己的内存管理方案。请尽可能的找出与内存有关的错误，并回答其注释中的附加题。

```
//  为什么B的delete还有第二个参数?
//  为什么D的delete却没有第二个参数?
//
class B {
public:
    virtual ~B();
    void operator delete  ( void*, size_t ) throw();
    void operator delete[]( void*, size_t ) throw();
    void f( void*, size_t ) throw();
};
class D : public B {
public:
    void operator delete  ( void* ) throw();
    void operator delete[]( void* ) throw();
};
void f()
```

```
{
    //  下面各个语句中，到底哪一个 delete 被调用了？为什么？
    //  调用时的参数是什么？
    //
    D* pd1 = new D;
    delete pd1;
    B* pb1 = new D;
    delete pb1;
    D* pd2 = new D[10];
    delete[] pd2;
    B* pb2 = new D[10];
    delete[] pb2;
    //  下面两个赋值语句合法吗？
    //
    B b;
    typedef void (B::*PMF)(void*, size_t);
    PMF p1 = &B::f;
    PMF p2 = &B::operator delete;
}
class X {
public:
    void* operator new( size_t s, int )
                    throw( bad_alloc ) {
        return ::operator new( s );
    }
};
class SharedMemory {
public:
    static void* Allocate( size_t s ) {
        return OsSpecificSharedMemAllocation( s );
    }
    static void  Deallocate( void* p, int i ) {
        OsSpecificSharedMemDeallocation( p, i );
    }
};
class Y {
public:
    void* operator new( size_t s,
                        SharedMemory& m ) throw( bad_alloc ) {
        return m.Allocate( s );
    }
    void  operator delete( void* p,
                           SharedMemory& m,
                           int i ) throw() {
        m.Deallocate( p, i );
    }
};
void operator delete( void* p ) throw() {
    SharedMemory::Deallocate( p );
}
void operator delete( void* p,
                      std::nothrow_t& ) throw() {
    SharedMemory::Deallocate( p );
}
```

[解答]

```
//  为什么 B 的 delete 还有第二个参数？
//  为什么 D 的 delete 却没有第二个参数？
//
class B {
public:
    virtual ~B();
```

```
    void operator delete  ( void*, size_t ) throw();
    void operator delete[]( void*, size_t ) throw();
    void f( void*, size_t ) throw();
};
class D : public B {
public:
    void operator delete  ( void* ) throw();
    void operator delete[]( void* ) throw();
};
```

附加题答案：这是出于个人喜好（preference）。两种都是正常的回收（译注：作者在这里用了“deallocation（去配）”一词,鄙人一律翻译为“回收”)函数,而不是 placement deletes(译注:关于 placement delete 和 placement new，可以阅读 Stanley Lippman 的《Inside The C++ Object Model（深度探索 C++对象模型)》一书中的 6.2 节：Placement Operator New 的语意)。

另外，这两个类都提供了 delete 和 delete[]，却没有提供相对应的 new 和 new[]。这是非常危险的。你可以试想，当更下层的派生类提供了它自己的 new 和 new[]时会发生什么！

```
void f()
{
    //  下面各个语句中，到底哪一个 delete 被调用了？为什么？
    //  调用时的参数是什么？
    //
    D* pd1 = new D;
    delete pd1;
```

这里调用了 D::operator delete(void*)。

```
    B* pb1 = new D;
    delete pb1;
```

这里调用 D::operator delete(void*)。因为 B 的析构函数（destructor）被声明成 virtual，所以 D 的析构函数（destructor）理所当然会被正常调用，但同时这也意味着，即使 B::operator delete()不声明成 virtual，D::operator delete()也必将被调用。

```
    D* pd2 = new D[10];
    delete[] pd2;
```

这里 D::operator delete[](void*)被调用。

```
    B* pb2 = new D[10];
    delete[] pb2;
```

这里的行为是不可预料的。C++语言要求，传递给 delete 的指针之静态类型必须与其之动态类型一样。关于这个问题的进一步谈论，可以参看 Scott Meyers 在《Effective C++》或者《More Effective C++》中有关“Never Treat ArraysP olymorphically”的部分（译注：这里指的应该是《More Effective C+》中的条款 3：绝对不要以多态方式处理数组)。

```
    //  下面两个赋值语句合法吗？
    //
    B b;
    typedef void (B::*PMF)(void*, size_t);
    PMF p1 = &B::f;
    PMF p2 = &B::operator delete;
}
```

第一个赋值语句没问题，但是第二个是不合法的，因为“void operator delete(void*,size_t)throw()”并不是 B 的成员函数，即使它被写在上面使其看上去很像是。这里有一个小伎俩需要弄清楚，即 new 和 delete 总是静态的，即使它们不被显式的声明为 static。总是把它们声明为 static 是个很好的习惯，这可以让所有阅读你代码的程序员们明白无误的认识到这一点。

```
class X {
public:
    void* operator new( size_t s, int )
                    throw( bad_alloc ) {
        return ::operator new( s );
    }
};
```

这会产生内存泄漏，因为没有相应的 placement delete 存在。下面的代码也是一样：

```
class SharedMemory {
public:
    static void* Allocate( size_t s ) {
        return OsSpecificSharedMemAllocation( s );
    }
    static void  Deallocate( void* p, int i ) {
        OsSpecificSharedMemDeallocation( p, i );
    }
};
class Y {
public:
    void* operator new( size_t s,
                        SharedMemory& m ) throw( bad_alloc ) {
        return m.Allocate( s );
    }
```

这里也产生内存泄漏，因为没有对应的 delete。如果在用这个函数分配的内存里放置对象的构造过程中抛出了异常，内存就不会被正常释放。例如：

```
    SharedMemory shared;
    ...
    new (shared) T; // if T::T() throws, memory is leaked
```

在这里，内存还无法被安全的删除，因为类中没有提供正常的 operator delete。这意味着，基类或者派生类的 operator delete（或者是那个全局的 delete）将会试图处理这个回收操作（这几乎肯定会失败，除非你替换掉周围环境中所有类似的 delete——这实在是一件繁琐和可怕的事情）。

```
    void  operator delete( void* p,
                           SharedMemory& m,
                           int i ) throw() {
        m.Deallocate( p, i );
    }
};
```

这里的这个 delete 毫无用处，因为它从不会被调用。

```
void operator delete( void* p ) throw() {
    SharedMemory::Deallocate( p );
}
```

这是一个严重的错误，因为它将要删除那些被缺省的::operator new 分配出来的内存，而不是被 SharedMemory::Allocate()分配的内存。你顶多只能盼来一个迅速的 core dump。真是见鬼！

```
void operator delete( void* p,
                      std::nothrow_t& ) throw() {
    SharedMemory::Deallocate( p );
}
```

这里也是一样，只是更为微妙。这里的 delete 只会在“new(nothrow)T”失败的时候才会被调用，因为 T 的构造函数（constructor）会带着一个异常来终止，并企图回收那些不是被 SharedMemory::Allocate()分配的内存。这真是又邪恶又阴险！

好，如果你回答出了以上所有的问题，那你就肯定是走在成为内存管理机制专家的光明大道上了。

条款 37：GotW#25 auto_ptr

难度：8/10

问题

考虑下面的代码：那些是好的，那些是安全的，那些是合法的， 那些是非法的？

```
auto_ptr<T> source() { return new T(1); }
void sink( auto_ptr<T> pt ) { }
void f() {
    auto_ptr<T> a( source() );
    sink( source() );
    sink( auto_ptr<T>( new T(1) ) );
    vector< auto_ptr<T> > v;
    v.push_back( new T(3) );
```

```
    v.push_back( new T(4) );
    v.push_back( new T(1) );
    v.push_back( a );
    v.push_back( new T(2) );
    sort( v.begin(), v.end() );
    cout << a->Value();
}
class C {
public:    /*...*/
protected: /*...*/
private:
    auto_ptr<CImpl> pimpl_;
};
```

答案

考虑下面的代码：那些是好的，那些是安全的，那些是合法的， 那些是非法的?

标准更新：这周〖这篇 GotW 发布的那一周〗，在 WG21/J16 于 Morristown NJ USA 的会议上，C++语言的最终草案投票表决通过。我们期待在下次会议上（Nice， March 1998）它是否能通过并成为一个 ISO 正式标准。

在 Jersey 的会议上，auto_ptr 被精化以满足委员会的要求。这期专刊纵览最终版本的 auto_ptr，以了解怎样和为什么被修改得更安全和很易用，以及怎样更好地使用它。

概要：

1.所有对 auto_ptr 的合法使用工作得和以前一样，除了你不能使用（必然反引用）空 auto_ptr。

2.对 auto_ptr 的危险的滥用成为非法。

感谢：感谢 Bill Gibbons、Greg Colvin、Steve Rumsby 和其他精心修改 auto_ptr 的人员。尤其是 Greg 多年来一直从事 auto_ptr 及相关类的修改工作以满足各方面的意见和要求，并应该得到公众对此工作的赞誉。

背景

auto_ptr 的最初动机是使得下面的代码更安全：

```
void f() {
    T* pt( new T );
    /*...more code...*/
    delete pt;
}
```

如果 f()从没有执行 delete 语句（因为过早的 return 或者是在函数体内部抛出了异常），动态分配的对象将没有被 delete，一个典型的内存泄漏。

使其安全的一个简单方法是用一个“灵巧”的类指针对象包容这个指针，并在其析构时自动删除此指针：

```
void f() {
    auto_ptr<T> pt( new T );
    /*...more code...*/
} // cool: pt's dtor is called as it goes out of
  // scope, and the allocated object is deleted
```

现在，这个代码将不再泄漏 T 对象，无论是函数正常结束还是因为异常，因为 pt 的析构函数总在退栈过程中被调用。类似地，auto_ptr 可以被用来安全地包容指针〖注意：关于安全的一个重要细节没有在本次提及，它见于 GotW #62 和《Exceptional C++》〗：

```
// file c.h
class C {
public:
    C();
    /*...*/
private:
    auto_ptr<CImpl> pimpl_;
};
// file c.cpp
C::C() : pimpl_( new CImpl ) { }
```

现在，析构函数不再需要删除 pimpl_指针，因为 auto_ptr 将自动处理它。我们将在结束时再次讨论这个例子。

源和移交

这就是它的出发点，但它干得更好。基于 Greg Colvin 的工作和经验，人们注意到如果定义了 auto_ptr 的拷贝函数，对其在函数间进行传递（作为参数或者返回值）非常有用。

这就是草案第二稿（Dec 1996）中的 auto_ptr 的实际工作方式，auto_ptr 的拷贝行为将所有权从源移交到目标。在拷贝后，只有目标 auto_ptr“拥有”这个指针并在适当的时候删除它，而源 auto_ptr 仍然包容同样的对象但并不“拥有”它于是也不删除它（否则会有二次删除问题）。你仍然可以使用这个指针同时通过拥有它或不拥有它的 atuo_ptr 对象。

例如：

```
void f() {
```

```
        auto_ptr<T> pt1( new T );
        auto_ptr<T> pt2;
        pt2 = pt1;  // now pt2 owns the pointer, and
                    // pt1 does not
        pt1->DoSomething(); // ok (before last week)
        pt2->DoSomething(); // ok
    } // as we go out of scope, pt2's dtor deletes the
      // pointer, but pt1's does nothing
```

让我们看第一部分代码：(注 1)

```
    auto_ptr<T> source() { return new T(1); }
    void sink( auto_ptr<T> pt ) { }
```

结论

	Before NJ	After NJ
Legal?	Yes	Yes
Safe?	Yes	Yes

这正是在 Taligent 人们所关注的：

- source()分配了一个新对象并用一个完全安全的方法将它返回给调用者，让调用者获得了指针的所有权。即使是调用者忽略了返回值（当然，你从不写忽略返回值的代码，对吧？），分配的对象仍然被安全地删除。

参见 GotW #21，它证明了为什么这是一个重要的习惯用法，因为通过 auto_ptr 包容它而返回有时是唯一的让函数强异常安全方法。

- sink()接受一个值传递的 auto_ptr，并接管它的所有权。当 sink()结束时，删除动作被执行（假设 sink()没有将所有权传递给别人）。因为 sink()函数在函数体中没有做任何事，调用“sink(a);”相当于写为“a.release();”。

下面的代码显示了 source()和 sink()的行为：

```
    void f() {
        auto_ptr<T> a( source() );
```

结论

	Before NJ	After NJ
Legal?	Yes	Yes
Safe?	Yes	Yes

此处，f()接管了从 source()返回的指针的所有权，并（忽略 f()代码后面部分的一些问题）在自动变量 a 超处生存范围时自动删除它。这很好，并显示了 auto_ptr 的值传递是可以工作的。

```
        sink( source() );
```

结论

	Before NJ	After NJ
Legal?	Yes	Yes
Safe?	Yes	Yes

由于这儿的 source()和 sink()的定义过于简单（比如，为空），这只不过是变了形的“delete new T(1);”。所以，它真的有用吗？好吧，假设 source 是个非空的厂函数而 sink()是个非空的消费函数，那么，是的，它有很大的意义并突然有规律地出现在真实世界的程序中。

```
        sink( auto_ptr<T>( new T(1) ) );
```

结论

	Before NJ	After NJ
Legal?	Yes	Yes
Safe?	Yes	Yes

再次地，一个变形的“delete new T(1)”，并且当 sink()是个非空的消费函数并接管了所指对象的所有权时，它是一个有用的习惯用法。

不能做的事情，以及为什么不能做

“那么，”你说道，“这太好了，明确地支持 auto_ptr 的拷贝是个好事。”是的，但它也使得你在最不期望的时候陷入水深火热之中，这也就是为什么反对草案 2 中的 auto_ptr 的形式的原因。这里有一个根本问题，我用黑体突出出来：

对于 auto_ptr，拷贝不是对等的。

它指出了在范型中使用 auto_ptr 时的重要影响：必须明白拷贝是不相等的。例如：

```
        vector< auto_ptr<T> > v;
```

结论

	Before NJ	After NJ
Legal?	Yes	No
Safe?	No	No

这是第一个问题，也是委员会想修正的事之一。简而言之，即使编译器甚至没有提示 warning，将 auto_ptr 放入容器也是不安全的。因为我们没有办法警告包容器：拷贝 auto_ptr 的语意是不同寻常的（传递了所有权，并改变了右侧对象的状态）。实际上，就我所知的绝大部分编译器都放过了这个代码，并甚至作为例子出现在它们的文档中。然而，它实际上是不安全的（并且，现在也是不合法的）。

问题是auto_ptr不满足放入容器的对象在类型上的要求,因为auto_ptr的拷贝并不相等。首先,并没有说vector必须对所包容的对象建一个“额外”的拷贝。当然，通常你期望vector不要做这个拷贝（因为它不是必须的，并且是低效的，并且基于竞争，卖方也不喜欢提供一个没必要的低效的库），但这并不一定如此，你也不能依赖它。

继续，因为开始发生错误：

```
v.push_back( new T(3) );
v.push_back( new T(4) );
v.push_back( new T(1) );
v.push_back( a );
```

（插一句：注意拷贝a到vector意味着'a'对象不在拥有它所携带的指针。过会儿再讲它。）

```
v.push_back( new T(2) );
sort( v.begin(), v.end() );
```

结论

	Before NJ	After NJ
Legal?	Yes	No
Safe?	No	No

这实在是个恶梦,这也是委员会要求修改的另外一个原因(草案2中的auto_ptr的形式被表决No的主要原因)。当你调用将拷贝元素的范型函数（如sort()）时，这写函数必须假设拷贝是相等的。例如，sort内部至少要保留一个“当前”元素的拷贝，而如果你想让它能工作于auto_ptr时，它必须有一个当前auto_ptr对象的拷贝（因此接管了它的所有权并放在一个临时auto_ptr对象中），并继续完成余下的工作（包括获得更多的现在不再拥有所有权的auto_ptr对象的拷贝以作为当前值),当排序完成时，临时对象被销毁，于是问题来了：序列中至少一个auto_ptr（作为当前值的拷贝的那个）不再拥有它包容的指针的所有权，实际上，这个指针已经被delete了！

草案2中的auto_ptr的问题是它没有提供保护——没有warning，什么都没有——以应付这样的错误代码。委员会要求auto_ptr或者去掉不同寻常的拷贝语义或者使得这些危险的代码无法编译，以便编译器能阻止你做危险的事情如构造一个auto_ptr的vector或试图对它进行排序。

挖掘不再拥有所有权的auto_ptr

```
    // (after having copied a to another auto_ptr)
    cout << a->Value();
}
```

结论

	Before NJ	After NJ
Legal?	Yes	No
Safe?	(Yes)	No

（我们假设a已经被拷贝了，但其指针没有被vector或sort删除。）在草案2下，它很正常，因为虽然不再拥有所有权，auto_ptr仍然留有一份拷贝，只不过不在自己超出生存范围时对其调用delete，因为它知道它不再拥有所有权的。

然而，现在，拷贝一个auto_ptr不但传递所有权，并且还将源auto_ptr置NULL。这特别阻止了任何人通过没有所有权的 auto_ptr 做任何事情。根据最终规则，这样做是非法的，其结果未定义（在大部分系统下通常是内核dump。）

简而言之：

```
void f() {
    auto_ptr<T> pt1( new T );
    auto_ptr<T> pt2( pt1 );
    pt1->Value(); // using a non-owning auto_ptr...
                  //  this used to be legal, but is
                  //  now an error
    pt2->Value(); // ok
}
```

这是对auto_ptr的常见用法的最终版本。

封装指针成员

```
class C {
public:    /*...*/
protected: /*...*/
private:
    auto_ptr<CImpl> pimpl_;
};
```

〖注意：关于安全的一个重要细节没有在本次提及，它见于GotW #62和《Exceptional C++》〗

结论

	Before NJ	After NJ
Legal?	Yes	Yes
Safe?	Yes	Yes

auto_ptr以前是，现在仍然是对指针成员的有效封装。它工作得非常象我们在开始处的“背景”一段中提起的例子，只是用不需要在C的析构函数中应付执行清理工作的麻烦替换了不需要在函数结束时执行清理工作的麻烦。

这儿仍然有一个警告，当然...就如同你仍然使用的是普通指针成员一样，你必须处理拷贝构造函数和赋值函数（哪怕是用申明它们为私有而不定义它们的方法来禁止它们），因为默认版本的行为是错误的。

最后的新闻："cont auto_ptr"的习惯用法

现在我们进入一个更深层次，你将发现一个有趣的技巧。在其它好处之外，精修后的 auto_ptr 也使得拷贝 const auto_ptr 非法。例如：

```
const auto_ptr<T> pt1( new T );
    // making pt1 const guarantees that pt1 can
    // never be copied to another auto_ptr, and
    // so is guaranteed to never lose ownership
auto_ptr<T> pt2( pt1 ); // illegal
auto_ptr<T> pt3;
pt3 = pt1;              // illegal
```

这个"const auto_ptr"的习惯用法可能成为常用技巧之一，而现在你可以说你知道它了。

我希望你喜欢这期的专刊，以献给 C++标准的 ISO 最终稿〖November 1997〗。

注 1.在最初的问题中，我忘了没有一个 T*到 auto_ptr<T>的转换函数，因为构造函数是"explicit"的。后面的代码修正了这一点。

10. 陷阱、易犯错误和反常作法（Traps, Pitfalls, and Anti-Idioms）

条款 38：GotW#11 对象等同问题（Object Identity）

难度：5 / 10

（"我到底是谁？"这个问题蕴含着如何确定两个指针是否真的指向同一个对象的问题）

[问题]

测试语句"this != &other" 是一个防止自赋值（self-assingment）的常见编码手法。那么，要到达这个"防止自赋值（self-assingment)"目的，现有的语句是否是必要的和/或充分的呢？为什么？或者为什么不？如果答案是否定的，你又如何进行修改呢？注意要区分"protecting against Murphy vs. protecting against Machiavelli"。

```
（* 译注：见本条款末尾）
T& T::operator=( const T& other ){
    if( this != &other ) {  // the test in question
        // ...
    }
    return *this;
}
```

[解答]

一个简短的解答：从技术上来看，它既不是必要的也不是充分的。在实践当中，它工作得颇好但也有可能在 C++标准中被修改。

- 论点：异常-安全（Murphy）

如果 operator=()是异常-安全的（exception-safe），那么你并不需要检查自赋值（self-assignment）。这里有两个效率上的不利因素：a）如果可以进行自赋值（self-assignment）检查的话，那就可以进行彻底的优化从而省略掉赋值操作；b）如果代码被写成异常-安全的（exception-safe），那么同时也使得代码损失了一部分效率（这也即是说"paranoia has a price principle"，意即"偏执狂也有权衡代价的原则"）。

- 不是论点的论点：多重继承

过去曾有人把多重继承与本条款讨论的问题联系起来，但实际上这个问题与多重继承没有任何关系。我们的讨论的是一个涉及到 C++标准怎样让你比较两个指针的技术性问题。

- 论点：运算符的重载（Machiavelli）

虽然一些类可能会提供它们自己的 operator&()，但是问题中对自赋值（self-assignment）的检查却很可能不如你所期望的那样运作，而是做一些完全不同的事情。用"protecting against Machiavelli"来意寓这种情况是因为，我们只能推测编写 operator=()的人也许大概知道他实现的类是不是也重载了 operator&()。

要注意，一个类可能也会提供一个 T::operator!=()，但这与本问题无关——它并不影响我们对自赋值（self-assignment）的检查，原因是：由于一个被重载的运算符至少有一个参数的类型必须是"类（class)"类型，因而我们不可能编写一个包含有两个 T*型参数的 T::operator!=()。

后记 1

下面是一个所谓的"笑话代码（code joke)"。信不信由你，竟然真有一些并无恶意但无疑是被误导了的代码编写者曾企图使用这样的代码：

```
T::T( const T& other ) {
  if( this != &other ) {
```

```
    // ...
  }
}
```

怎么样？你能第一眼就看出毛病吗？

（WQ 注：在 C++ Primer 中举过“class A a = a;”的例子，语法上无错，但行为上是错的，所以拷贝构造函数不必进行保护！）

后记 2

值得注意的是，还有另外一些情况，在这些情况当中，指针的比较也不是多数人光凭直觉就能考虑周到的。比如：

- James Kanze 指出，把指针比较成字符串的行为是未定义的。其原因（我还未见有人给出过）是 C++标准明确规定：允许编译器将字符串存放在重叠的内存区域以作为一种空间优化方案。
- 一般来说，虽然像诸如<，<=，>和>=等内建（built-in）运算符的运算结果在各种特定情况下（比如，同一个数组中的两个指向对象的指针）都有着良好的定义，但你还是不能用这些运算符对任意的指针进行比较。标准库的使用也是基于这种限制的，其中规定：less<>以及其它库函数必须给出各指针的次序（ordering），以使得我们可以创建，比方说，一个 key 为指针类型的 map，即 map<T*,U,less<T*>>。（译注：此处的 map 是 STL 的一部分，一个 map 包括 key 和 value 两部分，使用的时候需要#include <map>）

（* 译注：关于 Murphy 和 Machiavelli，侯捷先生在其系列书评的《C++/OOP 大系》中提到：“就我的英文程度而言，[Sutter99]（即《Exceptional C++》）读起来不若[Meyers96]（即《More Effective C++》）和[Meyers98]（即《Effective C++2/e》）那般平顺，原因是其中用了很多厘语、口语、典故。举个例子，Morphy law 是什麼，大家知道吗？（莫菲定律说：会出错的，一定会出错。）Machiavelli 又代表了什麼意思？（意大利政治家，以诈术闻名。）”

在这里，本条款的译者 kingofark 只能惭愧的说 kingofark 自己也没有完全理解本条款使用 Murphy 和 Machiavelli 两个词的用意。

Machiavelli：马基雅维利，尼克尔 1469-1527 意大利政治理论家，他的著作 君主论(1513 年)阐述了一个意志坚定的统治者不顾道德观念的约束如何获得并保持其权力；马基雅弗利，意大利新兴资产阶级思想政治家,历史学家。）

条款 39：GotW#19 自动转换（Automatic Conversions）

难度：4 / 10

（从一种型别到另一种型别的自动转换有时是极度方便的。本期 GotW 通过一个典型的例子说明为什么自动转换同时也是极度危险的。）

[问题]

标准的 string 不具有向 const char*进行自动转换的能力。它应该有吗？

* * * * *

[背景]：

把 string 作为 C 风格的 const char*来进行访问，经常是很有用的。实际上，string 也确有一个成员函数 c_str()专门用来完成这个任务——该函数返回 const char*。下面的代码体现出两者的区别：

```
string s1("hello"), s2("world");
strcmp( s1, s2 );                   // #1 (错误)
strcmp( s1.c_str(), s2.c_str() ) // #2 (ok)
```

如果能做到#1 就再好不过了，但#1 却是错误的，因为 strcmp 需要的是两个指针，而 string 和 const char*之间却不存在自动转换。#2 是正确的，但却因为要显式的调用 c_str()而使代码变长。如果我们能使用#1 难道不是更好一些吗？

[解答]

标准的 string 不具有向 const char*进行自动转换的能力。它应该有吗？

不，理所不该。避免编写自动转换几乎总是一个可取的办法，不管是以转换运算符编写还是以 single-argument non-explicit constructor（单引数非显式构造函数）编写。[注 1]

说隐式转换一般是不安全的，有两个主要原因：

a) 它会影响重载解析（overload resolution）；并且

b) 它会使“错误的”代码被不声不响的编译通过。

如果存在一个从 string 到 const char*的自动转换，那么这个转换动作将会在任何编译器认为需要的地方被调用。这便意味着，你会遇到各种各样的转换问题——与你在采用 non-explicit 转换构造函数时所遇到的问题是一样的。你将会很容易的写出看上去正确而实际上不正确的代码，这些代码原则上应该导致失败，但却可能由于古灵精怪的巧合而被编译通过并完成了与预期完全不同的操作。

有很多这样的例子。下面就是一个简单的例子：

```
string s1, s2, s3;
s1 = s2 - s3;   // 欧噢，或许原本是想要"+"
```

其中的减法毫无意义，应该是错误的。然而，如果存在从 string 到 const char*的隐式转换，那么这段代码就会被编译通过，因为编译器会不声不响的将两个 string 转换为 const char*，然后对两个指针施以相减操作。

摘自 GotW 编码标准，作为小结：

——避免编写转换运算符（Meyers96：24-31；Murray93：38，41-43；Lakos96：646-650）

[注 1]：这里我所关注的是隐式转换的普遍问题；其实关于「为什么 string class 不应该具有向 const char*的转换」这个问题，还有其它一些原因。这儿有几个进一步讨论该内容的参考：
Koenig97: 290-292
Stroustrup94 (D&E): 83

[部分参考]

Koenig97	Andrew Koenig. "Ruminations on C++" Addison-Wesley, 1997
Lakos96	John Lakos. "Large-Scale C++ Software Design" Addison-Wesley, 1996
Meyers96	Scott Meyers. "More Effective C++" Addison-Wesley, 1996
Murray93	Robert Murray. "C++ Strategies and Tactics" Addison-Wesley, 1993
Stroustrup94 (or D&E)	Bjarne Stroustrup. "The Design and Evolution of C++" Addison-Wesley, 1994

条款 40：GotW#22 对象的生存期（Object Lifetimes）（一）

难度：5 / 10

（“生存，还是灭亡……[译注：这是莎士比亚所著《哈姆雷特》中的名句]” 一个对象何时才算是真实存在的？这个问题用来考察一个对象何时才能被安全的使用。）

[问题]

评述下面的程序段。#2 处的代码使安全和/或合法的吗？请对你的回答做出解释。

```
void f() {
  T t(1);
  T& rt = t;
  // #1: 使用 t 或者 rt 做一些事情
  t.~T();
  new (&t) T(2);
  // #2: 使用 t 或者 rt 做一些事情
} // t 被再次销毁
```

[解答]

是的，#2 处的代码是安全且合法的（如果只考虑这部分代码的话），但：

a) 函数作为一个整体，它是不安全的，而且
b) 这样做是一个坏习惯。

[为什么#2 是安全的（如果只考虑这部分代码的话）？]

C++标准草案明确规定，允许这种代码出现。现场的析构和重构造（in-place destruction and reconstruction）不会使 rt 这个引用失效。（当然，你不能在 t.~T()与 placement new 之间使用 t 或 rt，因为在那段时期里不存在任何对象。我们还假设 T::operator&()没有被重载，即没有被用来做「返回对象之地址」以外的其它事情。）
我们之所以说“如果只考虑这部分代码的话，#2 就是安全的”，是因为 f()作为一个整体而言，可能不是异常安全的（exception-safe）：

[为什么函数是不安全的？]

如果在调用 T(2)的时候，T 的构造函数有抛出异常的可能，那么 f()就不是异常安全的。考虑其原因：如果 T(2)抛出异常，那么在原来’t’所在的内存区域中将不会有新的对象被构造，而在函数末尾 T::~T()仍然被正常调用（因为 t 是一个自动变量[automatic variable]），而且正如代码中的注释所述，“t 被再次销毁”。这即是说，’t’会被构造一次，却被销毁两次（呜呼呀）。这将导致容易产生无法预见的副作用，比如 core dumps。

[为什么这是个坏习惯？]

如果忽略异常安全性的问题，那么代码在这样的设定下恰好就能够正常工作，这是因为程序员此时知道被构造和销毁之对象的具体型别。这即是说，该对象是一个 T，并被作为一个 T 来被销毁和重新构造。

在实际的代码中，这种技术（即便真是编码所需）几乎不会被使用，并且这样做也是非常坏的习惯；原因是：如果其出现在成员函数中，那么其将会充满（有时难以捉摸的）危险：

```
void T::f( int i ) {
  this->~T();
  new (this) T(i);
}
```

现在这种技术还算安全吗？基本上来说，不安全。考虑下面的代码：

```
class U : /*...*/ public T { /* ... */ };
void f() {
  /*AAA*/ t(1);
  /*BBB*/& rt = t;
  // #1: 使用 t 或者 rt 做一些事情
  t.f(2);
  // #2: 使用 t 或者 rt 做一些事情
} // t 被再次销毁
```

如果”/*AAA*/”是”T”，那么#2 处的代码仍然可行，即使”/*BBB*/”不是”T”（ ”/*BBB*/”可能是 T 的基类）。

如果”/*AAA*/”是”U”（译注：而不是”T”），那么无论”/*BBB*/”是什么，都已经毫无悬念了。大概你所能期待的最好结果就是一个及时的 core dump，因为对 t.f()的调用将对象“切割（slices）”了。这里说的“切割”是指：t.f()用属于另一个不同型别的对象替换了原来的对象——这即是说函数使用了 T 而不是 U。即便是你意欲编写不可移植的代码，你也无法知晓「当原来 U 所在的内存区域被 T 对象之数据抹盖以后，其被作为 U 是否还可用？」。固然还是有情况尚佳的机率，但是请不要走到那个地步……这绝不是一次良好的实践。

本期 GotW 包含了一些基本的、有关现场析构和重构（in-place destruction and reconstruction）的安全性问题和切割问题。这为下期的“GotW 条款 23：对象生存期（第二部分）”作下铺垫。

条款 41：GotW #23 对象的生存期（Object Lifetimes）（二）

难度：6 / 10

（接着条款 22，本期条款考虑一个经常被推荐使用的 C++惯用法——它经常也是危险且错误的。）

[问题]

评述下面的惯用法（用常见的代码形式表达如下）：

```
T& T::operator=( const T& other ) {
    if( this != &other ) {
        this->~T();
        new (this) T(other);
    }
    return *this;
}
```

1. 代码试图达到什么样的合法目的？修正上述代码中所有的编码缺陷。

2. 假如修正了所有的缺陷，这种惯用法是安全的吗？对你的回答做出解释。如果其是不安全的，程序员又该如何达到预想的目标呢？

（参见 GotW 条款 22，以及 October 1997 C++ Report。）

[解答]

评述下面的惯用法（用常见的代码形式表达如下）：

```
T& T::operator=( const T& other ) {
    if( this != &other ) {
        this->~T();
        new (this) T(other);
    }
    return *this;
}
```

[小结][注 1]

这个惯用法经常被推荐使用，且在 C++标准草案中作为一个例子出现。[注 2]但其却具有不良的形式，而且——若要这么形容的话——恰恰是有害无益。请不要这样做。

1. 代码试图达到什么样的合法目的？

这个惯用法以拷贝构造（copy construction）操作来实现拷贝赋值（copy assignment）操作。这即是说，该方法试图保证「T 的拷贝构造与拷贝赋值实现的是相同的操作」，以避免程序员被迫在两个地方不必要的重复相同的代码。

这是一个高尚的目标。无论如何，它使编程更为简单，因为你不必把同一段代码编写两次，而且当 T 被改变（例如，给 T 增加了新的成员变量）的时候，你也不会像以前那样在更新了其中一个之后忘记更新另一个。

假如虚拟基类拥有数据成员，那么这个惯用法还是蛮有用的，因为若不使用此方法的话，数据成员在最好的情况下会被赋值数次，而在最坏的情况下则会被施以不正确的赋值操作。这听起来颇佳，但实际上却并无多大用处，因为虚拟基类其实是不应该拥有数据成员的。[注 3] 另外，既然有虚拟基类，那便意味着该类是为了用于继承而设计的——这又意味着：（正如我们即将看到的那样）我们不能使用这个惯用法，原因是它太具危险性。

修正上述代码中所有的编码缺陷。

上面的代码中包含一个可以修正的缺陷，以及若干个无法修正的缺陷。

[问题#1：它会切割对象]

如果 T 是一个带有虚拟析构函数（virtual destructor）的基类，那么”this->~T();”这一句就执行了错误的

操作。如果是对一个派生类的对象执行这个调用的话，这一句的执行将会销毁派生类的对象并用一个 T 对象替代。而这种结果几乎将肯定破坏性的影响后续任何试图使用这个对象的代码。(更多关于“切割（slicing)” 问题的讨论，参见 GotW 条款 22。)

特别要指出的是，这种状况将会使编写派生类的编码者们陷入人间地狱般的生活（另外还有其它一些关于派生类的潜在陷阱，见下面的叙述)。回想一下，派生的赋值运算符通常是基于基类的赋值操作编写的：

```
Derived&
Derived::operator=( const Derived& other ) {
    Base::operator=( other );
    // ...现在对派生成员进行赋值...
    return *this;
}
```

这样我们得到：

```
class U : /* ... */ T { /* ... */ };
U& U::operator=( const U& other ) {
    T::operator=( other );
    // ...现在对 U 成员进行赋值... 呜呼呀
    return *this;           //呜呼呀
}
```

正如代码所示，对 T::operator=()的调用一声不响的对其后所有的代码(包括 U 成员的赋值操作以及返回语句)产生了破坏性的影响。如果 U 的析构函数没有把它的数据成员重置为无效数值的话（译注：即可以编译运行通过)，这里将表现为一个神秘的、极难调试的运行期错误。

为了改正这个问题，可以调用"this->T::~T();"作为替代，这可以保证「对于一个派生类对象，只有其中的 T subobject 被替换（而不是整个派生类对象被切割从而被错误的转为一个 T 对象)」。这样做只是用一个更为微妙的危险替换掉了一个明显的危险，而这个替换方案仍然会影响派生类的编写者（见下面的叙述)。

2．假如修正了所有的缺陷，这种惯用法是安全的吗？

不，不安全。要注意：如果不放弃整个惯用法的话，下列任何一个问题都无法得到解决：

[问题#2：它不是异常安全的]

‘new’语句将会唤起 T 的拷贝构造函数。如果这个构造函数可以抛出异常的话（其实许多甚至是绝大部分的类都会通过抛出异常来报告构造函数的错误)，那么这个函数就不是异常安全的，因为其在构造函数抛出异常时会导致「销毁了原有对象而没有用新的对象替换上去」的情形。

与切割（slicing）问题一样，这个缺陷将会对后续的任何试图使用这个对象的代码产生破坏性影响。更糟糕的是，这还可能导致「程序试图将同一个对象销毁两次」的情况发生，因为外部的代码无法知晓这个对象的析构函数是否已经被运行过了。(参见 GotW 条款 22 中更多关于重复析构的讨论。)

[问题#3：它使赋值操作变得低效]

这个惯用法是低效的，因为赋值过程中的构造操作几乎总是涉及到比重置数值更多的工作。析构和重构在一起进行则更是增加了工作量。

[问题#4：它改变了正常的对象生存期]

这个惯用法破坏性的影响了那些依赖于正常的对象生存期之代码。特别是它破坏或干预了所有使用常见的“初始化就是资源获取（initialization is resource acquisition)”惯用法的类。

例如，若 T 在构造函数里获取了一个互斥锁（mutext lock）或者开启了数据库事务（database transaction)，又在析构函数里释放这个锁或者事务处理，那会发生什么呢？这个锁或者事务处理将会以不正确的方式被释放并在赋值操作中被重新获得——这一般来说会破坏性的影响客户代码（client code）和这个类本身。除了 T 和 T 的基类以外，如果 T 的派生类也依赖于 T 正常的生存期语义，它也会同样破坏性的影响这些派生类。

有人会说，“我当然决不会对一个在构造函数和析构函数中获取和释放互斥量的类使用这个惯用法了!”回答很简单：“真的吗？你怎么知道你使用的那些（直接或间接）的基类不这样做呢？”坦白的说，你经常是无法知晓这个情况的，你也绝不应该依赖那些工作起来似乎正常但却与对象生存期玩花招儿的基类。

这个惯用法的根本问题在于它搅浊了构造操作和析构操作的含义。构造操作和析构操作分别准确的对应于对象生存期的开始和结束，对象通常分别在这两个时刻获取和释放资源。构造操作和析构操作不是用来改变对象值的操作（实际上它们压根儿也不会改变对象的值，它们只是销毁原来的对象并替换上一个看起来一样、恰好拥有新数值的东西，其实这个新的东西与原来的对象根本就不是一回事儿)。

[问题#5：它可以对派生类产生破坏性影响]

用"this->T::~T();"作为替代语句解决了问题#1 之后，这个惯用法仅仅替换掉派生类对象中的 T subobject。许多派生类都可以如此正常工作，把它们的基类 subobject 换出换入，但有些派生类却可能不行。

特别要指出的是，有些派生类可对其基类的状态予以控制，如果在不知道此信息的情况下对这些派生类的基类 subobject 进行盲目修改（以不可见的方式销毁和重构一个对象当然也算作是一种修改)，那么这些派生类就可能导致产生失败。一旦赋值操作做了任何超出「一个“正常写入”型赋值运算符所应该做的操作」之额外操作，这个危险就会体现出来。

[问题#6：它依赖于不可靠的指针比较操作]

该惯用法完全依赖于"this != &other"测试。(如果你对此有疑问的话，请考虑自赋值的情形。)

其问题在于：这个测试并不保证你希望它保证的事情：C++标准保证「对指向同一个对象的多个指针的比较之结果必须是“相等(equal)” 」,但却并不保证「对指向不同对象的多个指针的比较之结果必须是“不相等(unequal)”」。如果这种情况发生，那么赋值操作就无法如愿完成。(关于"this != &other"测试的内容，参见 GotW 条款 11。)

如果有人认为这太钻牛角尖了，请参看 GotW 条款 11 中的简要论述：所有“必须”检查自赋值（self-assignment）的拷贝赋值操作都不是异常安全的。[注 4][注意：请看 Exceptional C++及其勘误表以得到更新的信息。]

另外还有一些能够影响客户代码和/或派生类的潜在危险（诸如虚拟赋值运算符的情形——这即使是在最好的情况下也还是多少有些诡异的），但到目前为止已经有足够多的内容用来演示该惯用法存在的严重问题了。

[那现在我们应该怎么做呢]

如果其是不安全的，程序员又该如何达到预想的目标呢？

用同一个成员函数完成两种拷贝操作（拷贝构造和拷贝赋值）的注意是很好的：这意味着我们只需在一个地方编写和维护操作代码。本条款问题中的惯用法只不过是选择了错误的函数来做这件事。如此而已。

其实，惯用法应该是反过来实现的：拷贝构造操作应该以拷贝赋值操作来实现，不是反过来实现。例如：

```
T::T( const T& other ) {
  /* T:: */ operator=( other );
}
T& T::operator=( const T& other ) {
  // 真正的工作在这里进行
  // (大概可以在异常安全的状态下完成，但现在
  // 其可以抛出异常，却不会像原来那样产生什么不良影响
  return *this;
}
```

这段代码拥有原惯用法的所有益处，却不存在任何原惯用法中存在的问题。[注 5] 为了代码的美观，你或许还要编写一个常见的私有辅助函数，利用其做真正的工作；但这也是一样的，无甚区别：

```
T::T( const T& other ) {
  do_copy( other );
}
T& T::operator=( const T& other ) {
  do_copy( other );
  return *this;
}
T& T::do_copy( const T& other ) {
  // 真正的工作在这里进行
  // (大概可以在异常安全的状态下完成，但现在
  // 其可以抛出异常，却不会像原来那样产生什么不良影响
}
```

[结论]

原始的惯用法中充满了缺陷，且经常是错误的，它使派生类的编写者过上人间地狱般的生活。我时常禁不住想把这个原始的惯用法贴在办公室的厨房里，并注明：“有暴龙出没。”

摘自 GotW 编码标准：

- **如果需要的话，请编写一个私有函数来使拷贝操作和拷贝赋值共享代码；千万不要利用「使用显式的析构函数并且后跟一个 placement new」的方法来达到「以拷贝构造操作实现拷贝赋值操作」这样的目的，即使这个所谓的技巧会每隔三个月就在新闻组中出现几次。**（也就是说，决不要编写如下的代码：）

```
T& T::operator=( const T& other )
{
    if( this != &other)
    {
        this->~T();             // 有害!
        new (this) T( other );  // 有害!
    }
    return *this;
}
```

[注 1]：这里我忽略一些变态的情形（例如，重载 T::operator&()，使其做出返回 this 以外的事情）。GotW 条款 11 提到一些有关情况。

[注 2]：在 C++标准草案中的那个例子意在演示对象生存期的规则，而不是要推荐一个好的现实用法（它不现实!）。下面给出草案 3.8/7 中的那个例子（处于空间的考虑做了微小的修改）以飨感兴趣的读者：

[例子:

```
struct C {
  int i;
  void f();
  const C& operator=( const C& );
};
const C& C::operator=( const C& other)
{
```

```
    if ( this != &other )
    {
      this->~C();     // '*this'的生存期结束
      new (this) C(other);
                      // 新的 C 型别的对象被创建
      f();            // 此处定义良好
    }
    return *this;
  }
  C c1;
  C c2;
  c1 = c2; //此处定义良好
  c1.f();  //此处定义良好; c1 指的是
           //  新的 C 型别的对象
```

--例子 完]

并不推荐实际使用该代码的进一步的证据在于：C::operator=()返回了一个 const C&而不单纯是 C&，这不必要的避免了这些对象在标准程序库之容器（container）中的可移植用法。

摘自 GotW 编码标准：

- **将拷贝赋值操作声明为"T& T::operator=(const T&)"**
- **不要返回 const T&，尽管这样做避免了诸如"(a=b)=c"的用法；这样做意味着：你无法出于移植性的考虑而将 T 对象放入标准程序库之容器——因为其需要赋值操作返回一个单纯的 T&（Cline95: 212; Murray93: 32-33）**

[注 3]：参见 Scott Meyers 的《Effective C++》

[注 4]：尽管你不能依赖于"this != &other"测试，但如果你为了通过优化处理排除已知的自赋值情形而这样做，则并没有错。如果它起作用的话，你便可以省掉一个赋值操作。当然，如果它不起作用的话，你的赋值运算符应该仍然以「对于自赋值而言是安全的」之方式来编写。关于使用这个测试作为优化手段，有人赞同也有人反对——但这超出了本期 GotW 的讨论范围。

[注 5]：的确，它仍然需要一个缺省的构造函数，并可能仍然不是最高效的；但要知道，你唯有利用初始化列表（initializer lists）才能得到最优的高效性（利用初始化列表即在构造过程中同时初始化成员变量，一气呵成，而不是分为先构造，再赋值两步来完成）。当然，这样做又要牺牲代码的公用性（commonality），而对此的权衡也超出了本期 GotW 的讨论范围。

11. 杂项主题（Miscellaneous Topics）

条款 42：GotW#1 变量的初始化（Variable Initialization）

难度：4 / 10

（想想看，有多少种将变量初始化的方法？ 千万要注意那些看上去很像“变量初始化”的东西。）

[问题]

下列四条语句有什么区别吗?

```
SomeType t = u;

SomeType t(u);

SomeType t();

SomeType t;
```

[解答]

我们按从下往上的顺序逐个考察四条语句：

- `SomeType t;`

  变量 t 被缺省构造函数 SomeType::SomeType()初始化。

- `SomeType t();`

这是一个骗局，因为这条语句看上去很像一个变量声明，而实际上却是一个函数声明；这个函数 t 没有参数并且返回类型为 SomeType。

- SomeType t(u);

这是直接初始化。变量 t 通过构造函数 SomeType::SomeType(u)被初始化。

- SomeType t = u;

这是拷贝初始化。变量 t 通过 SomeType 的拷贝构造函数(Copy Constructor)被初始化。(注意，这条语句虽然含有“=”，但仍然是一个初始化操作，而不是一个赋值操作，因为在这里，允许使用'='只是为了可以沿用 C 语言的语法，operator=是不会被调用的。)

[语义学参考]：如果 u 恰好也是 SomeType 类型，那么这条语句与“SomeType t(u);”是等同的，将调用 SomeType 的拷贝构造函数(Copy Constructor)。如果 u 是 SomeType 以外的其它类型，那么这条语句与“SomeType t(SomeType(u))”是等同的。可以看到，在语句“SomeType t(SomeType(u))”里，u 被转换成一个临时的 SomeType 对象，而 t 则是由此拷贝构造出来的。

[注意]：在这种情况下，编译器通常可以（但不是必须要）对其进行优化，适当的处理拷贝构造（Copy Construction)操作(一般是省略掉拷贝构造过程)。如果进行了优化，则一定要保证拷贝构造函数(Copy Constructor)的可达性。

[学习指导]：建议总是使用“SomeType t(u)”的形式， 一来是因为只要可以用“SomeType t = u”的地方也同样可以它；二来是因为它还有一些其它的优点，比如支持多个参数等。

条款 43：GotW#6 正确使用 const（Const-Correctness）

难度：6 / 10

（总是尽可能的使用 const，但也不要用得太过分而造成滥用。哪些地方应该用 const，哪些地方不应该用？这里我们列举一些或明显或不明显的例子。）

[问题]

Const 是编写“安全”代码的一个强有力的工具，还有利于编译器的优化处理。你应该尽可能的使用它……但是，“尽可能的”到底是什么意思？

请不要评价这个程序的好坏，也不要改变它的结构；这个程序是为了说明本条款的问题而故意精心设计的。你只要在适当的地方对其添加或者删除“const”(包括各种变体和相关的关键字)就可以了。(附加题：哪些地方的 const 用法造成了程序有无法预料的结果或者无法编译通过的错误？)

```
class Polygon {
public:
  Polygon() : area_(-1) {}
  void AddPoint( const Point pt ) {
    InvalidateArea();
    points_.push_back(pt);
  }
  Point GetPoint( const int i ) {
    return points_[i];
  }
  int GetNumPoints() {
    return points_.size();
  }
  double GetArea() {
    if( area_ < 0 ) // 如果还没有被计算和保存，
      CalcArea();     // 那么现在就开始。
    return area_;
    }
private:
  void InvalidateArea() { area_ = -1; }
  void CalcArea() {
    area_ = 0;
```

```
    vector<Point>::iterator i;
    for( i = points_.begin(); i != points_.end(); ++i )
      area_ += /* some work */;
  }
  vector<Point> points_;
  double        area_;
};
Polygon operator+( Polygon& lhs, Polygon& rhs ) {
  Polygon ret = lhs;
  int last = rhs.GetNumPoints();
  for( int i = 0; i < last; ++i ) // 连接
    ret.AddPoint( rhs.GetPoint(i) );
  return ret;
}
void f( const Polygon& poly ) {
  const_cast<Polygon&>(poly).AddPoint( Point(0,0) );
}
void g( Polygon& const rPoly ) {
  rPoly.AddPoint( Point(1,1) );
}
void h( Polygon* const pPoly ) {
  pPoly->AddPoint( Point(2,2) );
}
int main() {
  Polygon poly;
  const Polygon cpoly;
  f(poly);
  f(cpoly);
  g(poly);
  h(&poly);
}
```

[解答]

```
class Polygon {
public:
  Polygon() : area_(-1) {}
  void AddPoint( const Point pt ) {
    InvalidateArea();
    points_.push_back(pt);
  }
```

1. 因为 Point 对象采用的是传值方式，所以把它声明为 const 几乎没有什么油水可捞，不会对性能有显著影响。

```
  Point GetPoint( const int i ) {
    return points_[i];
  }
```

2. 同 1，const 值传递没有多大意义，反而容易引起人误解。

3. 这个成员函数应该定义成一个 const 成员函数，因为它不改变对象的状态。

4. （本要点是一个有争议的提法）如果该函数的返回类型不是一个内建（built-in）类型的话，通常应该将其返回类型也声明为 const。这样做有利于该函数的调用者，因为它使得编译器能够在函数调用者企图修改临时量的时候产生一个错误信息，从而达到保护目的（例如， “Poly.GetPoint(i) = Point(2, 2);”，等等。诚然，如果真的想要修改临时量，那么首先就应该让 GetPoint()返回引用（return-by-reference）而不是返回值（return-by-value）。然而在后面的叙述中我们又将看到，在用于 operator+()中的 const Polygon 对象时，让 GetPoint()返回 const 值或者 const 引用又是很有用的。）。

[作者记：Lakos(pg.618)对返回 const 值提出异议。他认为，对于内建（built-in）类型也返回 const 值不但没有什么实际意义的，反而还会影响模板的实体化过程（instantiation）。]

[学习指导]：在函数内对非内建（non-built-in）的类型采用值返回（return-by-value）的方法时，最好让函数返回一个 const 值。

```
  int GetNumPoints() {
```

```
    return points_.size();
  }
```

5. 还是那句话，函数本身应该被声明为 const。

（注意在这里不应该返回 const int，因为 int 本身已经是一个右值类型；加上 const 以后反而会影响模板的实体化过程（instantiation），使代码变得容易让人糊涂，引人误解，甚至让人消化不良也有可能。）

```
double GetArea() {
if( area_ < 0 ) //如果还没有被计算和保存，
  CalcArea();     //那么现在就开始。
return area_;
}
```

6. 尽管这个函数修改了对象的内部状态，但它还是应该被声明为 const，因为被修改对象的可见（observable）状态没有发生变化（我们所做的是一些隐蔽的、不可告人的事情，但这是一个实现中的细节，即这个对象从逻辑上讲还是 const）。这意味着，area_应该被声明成 mutable。如果你的编译器目前还不支持 mutable 关键字的话，可以对 area_采取 const_cast 操作以作为替代方案（最好还能在这里写一个注释，以便在可以使用 mutable 关键字的时候把这个 cast 操作替换掉。），但记住一定要让函数本身被声明为 const。

```
private:
  void InvalidateArea() { area_ = -1; }
```

7. 尽管这一观点也是备受争议，我还是坚持认为，即使仅仅只是出于一致性的考虑，也还是应该把这个函数声明为 const。（当然我不得不承认，从语义学的角度上讲，这个函数只会被非 const 的函数调用，因为毕竟其目的只是想在对象的状态改变时让保存的 area_值无效。）

```
void CalcArea() {
  area_ = 0;
  vector<Point>::iterator i;
  for( i = points_.begin(); i != points_.end(); ++i )
    area_ += /* some work */;
}
```

8. 这个成员函数绝对应该是 const 才对。不管怎么说，它至少会被另外一个 const 成员函数即 GetArea()调用。

9. 既然 iterator 不会改变 points_集的状态，所以这里应该是一个 const_iterator。

```
vector<Point> points_;
  double        area_;
};
Polygon operator+( Polygon& lhs, Polygon& rhs ) {
```

10. 显然，这里应该通过 const 引用（reference）进行传递。

11. 还是那句话，返回值应该是 const 的。

```
  Polygon ret = lhs;
  int last = rhs.GetNumPoints();
```

12. 既然“last”也从不会被改变，那么也应该为“const int”型。

（这也是 GetPoint()——不管它返回的是 const 值还是 const 引用——必须是一个 const 成员函数的原因。）

```
  return ret;
}
void f( const Polygon& poly ) {
  const_cast<Polygon&>(poly).AddPoint( Point(0,0) );
```

附加题的要点：如果被引用的对象被声明为 const，那么这里的结果将是未定义的（如同下面要讲的 f(cpoly)一样）。事实是，其参数并不真是 const 的，所以千万不要把它声明为 const！

```
}
void g( Polygon& const rPoly ) {
  rPoly.AddPoint( Point(1,1) );
}
```

13. 这里的 const 毫无作用，因为无论如何一个引用（reference）是不可能被改变，使其指向另一个对象的。

```
void h( Polygon* const pPoly ) {
  pPoly->AddPoint( Point(2,2) );
}
```

14. 这里的 const 也不起作用，但其原因与 13 中的不同：这次是因为你对指针使用传值方式，这与上面讲到的传递一个 const int 参数一样毫无意义。

（如果你在对附加题的解答中提到这里的函数会产生编译错误的话……不好意思，它们是合法的 C++用法。也许你还考虑过把 const 放在&或者*的左边，但那只会使函数体本身不符合 C++用法。）

```
int main() {
 Polygon poly;
 const Polygon cpoly;
 f(poly);
```

这儿很好，没什么问题。

```
   f(cpoly);
```

在这里，当 f()企图放弃 const 属性从而修改其参数的时候，会产生不确定的结果。

```
   g(poly);
```

这一语句没问题。

```
   h(&poly);
```

这一句也没问题。

```
}
```

好了，终于完了！现在得到了正确的代码版本（当然，只改正了有关 const 的错误，而不管其不良的编码风格）：

```
class Polygon {
public:
    Polygon() : area_(-1) {}
    void  AddPoint( Point pt )       { InvalidateArea();
                                       points_.push_back(pt); }
    const Point GetPoint( int i ) const  { return points_[i]; }
    int         GetNumPoints() const { return points_.size(); }
    double GetArea() const {
        if( area_ < 0 ) // 如果还没有进行计算和保存，
            CalcArea();     //那么现在就开始。
        return area_;
    }
private:
    void InvalidateArea() const { area_ = -1; }
    void CalcArea() const {
        area_ = 0;
        vector<Point>::const_iterator i;
        for( i = points_.begin(); i != points_.end(); ++i )
            area_ += /* some work */;
    }
    vector<Point>  points_;
    mutable double area_;
};
const Polygon operator+( const Polygon& lhs,
                         const Polygon& rhs ) {
    Polygon ret = lhs;
    const int last = rhs.GetNumPoints();
    for( int i = 0; i < last; ++i ) // 连接
        ret.AddPoint( rhs.GetPoint(i) );
    return ret;
}
void f( Polygon& poly ) {
    poly.AddPoint( Point(0,0) );
}
void g( Polygon& rPoly ) { rPoly.AddPoint( Point(1,1) ); }
void h( Polygon* pPoly ) { pPoly->AddPoint( Point(2,2) ); }
int main() {
    Polygon poly;
    f(poly);
```

```
    g(poly);
    h(&poly);
}
```

条款 44：GotW#17 类型转换（Casts）

难度：6 / 10

（你对 C++转型了解多少？适当的使用它可以极大的提高代码的可靠性。）

[问题]

标准 C++中新风格的转型与旧风格的 C 转型相比，具有更强大的功能和安全性。你对它了解多少？本条款中使用下列类和全局变量：

```
class  A             { /*...*/ };
class  B : virtual A { /*...*/ };
struct C : A         { /*...*/ };
struct D : B, C      { /*...*/ };
A a1; B b1; C c1; D d1;
const A a2;
const A& ra1 = a1;
const A& ra2 = a2;
char c;
```

1. 下列哪一种新风格的转型不能与 C 中的转型相对应？

- const_cast
- dynamic_cast
- reinterpret_cast
- static_cast

2. 对于下列每一个 C 中的转型语句，写出相应的新风格转型语句。其中哪一个语句如果不以新风格编写的话就是不正确的？

```
void f() {
  A* pa; B* pb; C* pc;
  pa = (A*)&ra1;
  pa = (A*)&a2;
  pb = (B*)&c1;
  pc = (C*)&d1;
}
```

3. 评判下列每一条 C++转型语句的编写风格和正确性。

```
void g() {
  unsigned char* puc = static_cast<unsigned char*>(&c);
  signed char* psc = static_cast<signed char*>(&c);
  void* pv = static_cast<void*>(&b1);
  B* pb1 = static_cast<B*>(pv);
  B* pb2 = static_cast<B*>(&b1);
  A* pa1 = const_cast<A*>(&ra1);
  A* pa2 = const_cast<A*>(&ra2);
  B* pb3 = dynamic_cast<B*>(&c1);
  A* pa3 = dynamic_cast<A*>(&b1);
  B* pb4 = static_cast<B*>(&d1);
  D* pd = static_cast<D*>(pb4);
  pa1 = dynamic_cast<A*>(pb2);
  pa1 = dynamic_cast<A*>(pb4);
  C* pc1 = dynamic_cast<C*>(pb4);
  C& rc1 = dynamic_cast<C&>(*pb2);
}
```

[解答]

1. 下列哪一种新风格的转型不能与 C 中的转型相对应？

只有 dynamic_cast 不能与 C 的转型相对应。其它的新风格转型都能与 C 中的旧风格转型相对应。

2. 对于下列每一个 C 中的转型语句，写出相应的新风格转型语句。其中哪一个语句如果不以新风格编写的话就是不正确的？

```
void f() {
  A* pa; B* pb; C* pc;
  pa = (A*)&ra1;
```

应该使用 const_cast：const_cast<A*>(&ra1);

```
  pa = (A*)&a2;
```

这一句无法以新风格的转型表达。最接近的方案是使用 const_cast，但 a2 是一个 const object，语句执行的结果是未定义的。

```
  pb = (B*)&c1;
```

应该使用 reinterpret_cast：pb=reinterpret_cast<B*>(&c1);

```
  pc = (C*)&d1;
}
```

这一转型在 C 中是错误的。而在 C++中，并不需要转型：pc=&d1;

3. 评判下列每一条 C++转型语句的编写风格和正确性。

首先要注意：我们并不知道本条款中给出的类是否拥有虚拟函数；如果涉及转型的那些类并不拥有虚拟函数，那么下述所有对 dynamic_cast 的使用都是错误的。在下面的讨论中，我们假设所有的类都拥有虚拟函数，从而使所有的 dynamic_cast 用法都合法。

```
void g() {
  unsigned char* puc = static_cast<unsigned char*>(&c);
  signed char* psc = static_cast<signed char*>(&c);
```

错误：对两条语句我们都必须使用 reinterpret_cast。这一开始或许会使你感到吃惊；这样做的原因是，char、signed char 以及 unsigned char 是三个互不相同、区别开来的型别。尽管它们之间存在着隐式转换，它们也是互无联系的，因而指向它们的指针也是互无联系的。

```
  void* pv = static_cast<void*>(&b1);
  B* pb1 = static_cast<B*>(pv);
```

这两句都不错，但第一句中的转型是不必要的，因为本来就有从一个对象指针到 void*的隐式转型动作存在。

```
  B* pb2 = static_cast<B*>(&b1);
```

这一句不错，但其转型也是不必要的，因为其引数（argument）已经是一个 B*。

```
  A* pa1 = const_cast<A*>(&ra1);
```

这一句是合法的，但是使用转型来去掉 const-ness（常量性）是潜在的不良风格的体现。在大部分情况下，即当你因合理的缘由而想要去掉指针或引用的 const-ness(常量性)时，这都涉及到某些类成员，并通常会使用 mutable 关键字来完成。请参看 GotW#6 了解更多关于 const-correctness 的讨论。

```
  A* pa2 = const_cast<A*>(&ra2);
```

错误：如果该指针被用来对对象施行写操作，那么就会产生未定义行为；因为 a2 是一个 const object。要明白其原因，可以试想如果一个编译器了解到“a2 是作为 const object 而被创建的”这个情况，并出于优化的考虑而将其存放在只读存储区，会发生什么事情。很明显，想通过转型而去掉这样一个对象的 const 属性是危险的。

注意：我并没有举例显示如何使用 const_cast 把一个 non-const 指针转型为一个 const 指针。因为这样做是多此一举；将一个 non-const 指针赋值给一个 const 指针，这本来就是合法的。我们只需要使用 const_cast 做相反的操作。

```
 B* pb3 = dynamic_cast<B*>(&c1);
```

错误（当你企图使用 pb3 时发生）：这一句会将 pb3 设置为 null，因为 c1 不是一个（IS-NOT-A）B（因为 C 不是以 public 方式派生自 B 的，且实际上压根儿就不是派生自 B 的）。这里唯一合法可用的转型就是 reinterpret_cast，但使用它也几乎总是很龌龊的。

```
  A* pa3 = dynamic_cast<A*>(&b1);
```

错误：这一句是非法的，因为 b1 不是一个（IS-NOT-A）A（因为 B 不是以 public 方式派生自 A 的，而是以 private 方式）。

```
  B* pb4 = static_cast<B*>(&d1);
```

这一句不错，但也没必要做转型，因为 derived-to-base（由派生类到基类）的指针转换可以被隐式的完成。

```
  D* pd = static_cast<D*>(pb4);
```

这一句不错。如果你原先认为这里需要的是 dynamic_cast 的话，这或许会使你感到吃惊。其原因是，当目标已知的时候，向下转型（downcast）可以是静态的，此时要注意：你这样等于是在告诉编译器，你知道“被指针所指的正是那种型别”这个事实。如果你错了，那么这个转型将无法告知你已经出现的问题（dynamic_cast 在转型失败时就能返回一个 null pointer 以告知你出现了问题），于是你此时至多也只能得到各种不同的运行期错误以及/或者程序崩溃。

```
  pa1 = dynamic_cast<A*>(pb2);
  pa1 = dynamic_cast<A*>(pb4);
```

这两句看起来很相似。两句都试图使用 dynamic_cast 来把 B*转换为 A*。然而，第一个是错误的而第二个是正确的。

原因是：正如前面所述，你不能使用 dynamic_cast 把一个指向 B 对象（这里 pb2 指向对象 b1）的指针转换为指向 A 对象的指针，因为 B 是以 private 方式从 A 进行继承的，不是以 public 方式。然而第二句中的转型是成功的，这是因为 pb4 指向对象 d1，而 D（通过 C）将 A 作为一个间接的 public base class，从而让 dynamic_cast 可以沿着 B*-->D*-->C*-->A*的路径在继承层次结构中进行转型。

```
  C* pc1 = dynamic_cast<C*>(pb4);
```

这一句也不错，其原因与上面的一样：dynamic_cast 可以穿越继承层次进行交叉转型（cross-cast），因此这一句是合法的并可以成功执行。

```
 C& rc1 = dynamic_cast<C&>(*pb2);
```

最后这一句是错的……因为*pb2 并不真的就是一个 C，dynamic_cast 会抛出一个 bad_cast 异常来报告失败。

为什么？因为 dynamic_cast 可以在指针转型（pointer cast）失败时返回 null，但由于没有 null reference 一说，因此当一个引用转型（reference cast）失败时便无法返回 null reference。除了抛出一个异常以外，没有别的方法来报告错误了——标准的 bad_cast 异常类也就是因此而来的。

条款 45：GotW#26 Bool

难度：7/10

我们有需要基本的 bool 类型吗？，现存的语言为什么不是仅仅的仿真它，本期将给你答案。

问题：

自从 ARM.[1]发布以来，除了 wchar_t 之外（wchar_t 在 C 里是用 typedef 实现的），bool 是唯一添加到 C++里的基本数据类型；如果不添加这个基本类型，能用已有的数据类型达到相同的效果吗？如果能，请给出一个实现，如果不能，请指出各种可能的实现方法都有哪些缺陷？

解决方法：

自从 ARM.[1]发布以来，除了 wchar_t 之外（wchar_t 在 C 里是用 typedef 实现的），bool 是唯一添加到 C++里的基本数据类型；如果不添加这个基本类型，能用已有的数据类型达到相同的效果吗？如果能，请给出一个实现，如果不能，请指出各种可能的实现方法都有哪些缺陷？

回答是：不能，bool 明确地添加到 C++作为基本类型（并保留 true 与 false 关键字），因为它们（bool 型和 true, false）没法通过现有的语言特性准确实现。

如果不是，将给出同等实现基本类型的几个潜在的原因。

4 种主要的实现方法：

第一种：Typedef(评价:8.5/10)

这种方法用"typedef <something> bool;",典型如：

```
typedef int bool;
const bool true  = 1;
const bool false = 0;
```

这种解决方法不错，但是不允许重载 bool,例如：

```
// file f.h
void f( int  ); // ok
void f( bool ); // ok, 重新声明相同的函数
// file f.cpp
void f( int  ) { /*...*/ }   // ok
void f( bool ) { /*...*/ }   // error, 重定义
```

另一个问题是跟这相同的代码：

```
void f( bool b ) {
    assert( b != true && b != false );
}
```

于是可以看出第一种实现不够好！

第二种方法：#define(评价：0/10)

这种方法用"#define bool <something>",典型如：

```
#define bool  int
#define true  1
#define false 0
```

这种方法是有害的，这不仅仅与上面的第一种方法有相同的问题，而且通常破坏了#defines,例如：它缺乏自定义，且当有人在尝试使用这个库时已经有了变量名字'false';现在就可以从基本类型看到它们的差异。

尝试用预处理程序来模拟这种类型是很坏的主意。

第三种方法：Enum(评价：9/10)

用这种方法去产生"enum bool",典型如：

```
enum bool { false, true };
```

这种方法比第一种方法好几分，在这种方法，它允许重载(第一种方法的主要问题),但在条件表达式不允许自动类型转换(第一种方法是可以的)：

```
bool b;
```

```
b = ( i == j );
```
这是错误的，因为 int 不能隐含地转换成 enums

第四种方法：Class(评价：9/10)

这是面向对象语言，对吗？，所以为什么不写个类，典型如：

```
class bool {
public:
    bool();
    bool( int );      // 在条件表达式允许转达换
    operator=( int ); //
    //operator int();   // 有问题!
    //operator void*(); // 有问题!
private:
    unsigned char b_;
};
const bool true ( 1 );
const bool false( 0 );
```

除了转换操作有问题之外就可以工作了，这些问题的原因：

- 由于自动类型转达换，重载解决了 bool 的冲突，(尤其是基本类型的转换)
- 如果使用自动类型转换，bool 型的数据会影响到函数重载，就像其他任何一个类里的隐式构造函数和（或）自动类型转换一样（尤其是基本类型的转换）。
- 不能转换与 int 或 void*相似的类型，bool 在条件中不能作测试，例如：
- 如果不使用自动类型转换，那么 bool 类型将无法使用在条件判断中

```
bool b;
/*...*/
if( b ) // 错误，不能转换与 int 或 void*相似的类型
{       //
    /*...*/
}
```

这种方法抓住了 22 种情形：我们必须选择一个或其它的，两种方法皆让我们

这是一个经典的所谓的 Catch-22 困境：我们必须也只能选择其中的一个，但没有一种情况可以满足我们的需要：使这个类表现得和基本类型 bool 一样。

总结：

- typedef ... bool 不允许重载。
- #define bool 不允许重载且通常破坏了#define
- enum bool 允许重载但在条件表达式不能进行自动类型转换(相似于"b = (i == j);")。
- bool 类允许重载但不能让 bool 对象在条件中作测试，除非它能提供自动转换到基本类型，可它通常以破坏了类型自动转换。

对极了，我们真的很需要基本成的 bool 类型，最后，还有一件事情（和重载有关的），就是，对于所有的方式（除了在第四种方式下有可能)我们都无法指明那个条件判断表达式是一个 bool 类型。(我想作者要表达的意思是，对于各种模拟 bool 的方式，我们都无法把 bool 当作一种真正意义上的数据类型，并使用它来重载函数，考虑这样的调用：func(i == j); 在被当作参数去寻找匹配的函数之前，你根本无法改变这个条件判断表达式运算的结果的类型，其实我觉得关于这点他在前面已经讲得很清楚了，不知道为什么要称作 one more thing)

备注：

注 1： M. Ellis M and B. Stroustrup. **The Annotated C++ Reference Manual** (Addison-Wesley, 1990).

条款 46：GotW#27 转呼叫函数（Forwarding Functions）

难度：3 / 10

怎样将转呼叫函数写得最好？原本答案很简单，但我们已经知道 C++语言近来发生了微妙的变化。

问题

转呼叫函数对于将任务传递给其它函数或对象时很有用，尤其当它们被设计得很高效时。

评论一下下面这个转呼叫函数。你试图修改它吗？如果是的话，怎么来？

```
// file f.cpp
#include "f.h"
/*...*/
bool f( X x ) {
    return g( x );
}
```

（说明：本次 GotW 的目的之一是阐明在 July [1997]于 London 加入到 C++语言中的一个微妙改进所造成的后果。）

解答

转呼叫函数对于将任务传递给其它函数或对象时很有用，尤其当它们被设计得很高效时。

关键点是：效率。

评论一下下面这个转呼叫函数。你试图修改它吗？如果是的话，怎么来？

```
// file f.cpp
#include "f.h"
/*...*/
bool f( X x ) {
    return g( x );
}
```

有两个主要改进可使得这个函数更高效。第一个应该总被采用，第二个需要权衡。

1.传参时使用传 const 的引用代替传值

“这不会造成混乱吗？”你可能会问。不，它不会，至少在这种情况下。直到最近，C++语言才规定：因为编译器可以确保参数 x 除了被传递给 g()外没有被其它地方使用，编译器可以将 x 完全优化掉。例如，这样的代码：

```
X my_x;
f( my_x );
```

编译器可以：

a)产生一个 my_x 的拷贝供 f()使用（就是 f()的代码体中的形参 x），然后将这个拷贝传给 g()；或者

b)直接将 my_x 传给 g()而不生成拷贝，因为它注意到这个额外的拷贝除了作 g()的参数外根本没被使用。

后者更高效，不是吗？这是编译器试图作的优化，不是吗？

是的，是的，但只到 July 1997 的 London 会议。在那次会议上，“限制编译器作这种取消额外拷贝的优化”的提案得到了更多的支持。〖注 1〗编译器唯一可以取消额外拷贝构造的地方是“返回值优化”（在你的 C++宝典中查询细节吧）和“临时对象”。

这意味着，象 f 这样的转呼叫函数，编译器被要求产生两份拷贝。既然我们（作为 f 的作者）知道这个额外的拷贝不是必须的，我们应该按照通常的办法将 x 申明为 const X&型的参数。

（注意：如果我们一直就是这么做的，而不是依赖于知道编译器被允许做些什么，那么，这个规则的变化不会对我们造成任何影响。这就是一个“简单就是美”例子——尽可能避开语言的细枝末节，别耍小聪明。）

2.函数内联

这个需要权衡。要之，默认将所有函数都实现为外联，有选择地将确实需要内联以提高效率的函数实现为内联。当你将函数内联时，积极面是你避免了对 f 函数的调用的额外开销。

消极面是内联 f 暴露了 f 的实现，并使得用户的代码依赖于此实现，当 f 被改变时，所有的用户代码都必须被重编译。更严重的是，用户代码现在至少需要知道函数 g()的原型，这有点恶心，因为用户根本没有直接调用函数 g，原本可以根本不需要知道它的原型的（至少，从我们的例子上，是这样的）。于是，如果 g()自己发生了变化，接受其它类型的其它参数时，用户的代码将变得也需要知道这些类型的申明。

内联和非内联都可以。必须在优缺点间进行权衡，取决于 f 现在是怎样被使用的以及使用的广泛程度，和将来可能变为怎样被使用的以及使用的广泛程度。

GotW 给出的代码规范：

- **传参时，用传 const 的引用来代替传值**
- **避免函数内联，除非 profiler 告诉你有这个必要（程序员在猜测效能瓶颈点方面是很不准的）**

注 1：这个改进是必要的，它避免了编译器未经允许地省略拷贝构造时带来问题，尤其当拷贝构造有副作用时。很多时候，代码需要计算对象的拷贝数目。

条款 47：GotW#12 控制流（Control Flow）

难度：6 / 10

（你到底对 C++代码的执行顺序了解多少？用下面的问题来考查你的学识吧！）

[问题]

所谓“妖藏巨细（The devil is in the details.）。”，现在就请你从下面的（多少是有些故意设计的）代码中找出尽可能多的有关控制流（control flow）的毛病。

```
#include <cassert>
#include <iostream>
#include <typeinfo>
#include <string>
using namespace std;
//  下面的几句来自其它的头文件
char* itoa(int value, char* workArea, int radix );
```

```
extern int fileIdCounter;
 //  使类的不变量检查自动化的辅助函数
 template<class T>
inline void AAssert( T& p ) {
    static int localFileId = ++fileIdCounter;
    if( !p.Invariant() ) {
        cerr << "Invariant failed: file " << localFileId
             << ", " << typeid(p).name()
             << " at " << static_cast<void*>(&p) << endl;
        assert( false );
    }
}
 template<class T>
class AInvariant {
public:
    AInvariant( T& p ) : p_(p) { AAssert( p_ ); }
    ~AInvariant()              { AAssert( p_ ); }
private:
    T& p_;
};
#define AINVARIANT_GUARD AInvariant<AIType>  \
                         invariantChecker( *this )
 //-----------------------------------------------
class Array : private ArrayBase, public Container {
    typedef Array AIType;
public:
    Array( size_t startingSize = 10 )
      : Container( startingSize ),
        ArrayBase( Container::GetType() ),
        used_(0),
        size_(startingSize),
        buffer_(new char[size_])
    {
        AINVARIANT_GUARD;
    }
     void Resize( size_t newSize ) {
        AINVARIANT_GUARD;
        char* oldBuffer = buffer_;
        buffer_ = new char[newSize];
        memset( buffer_, ' ', newSize );
        copy( oldBuffer, oldBuffer+min(size_,newSize),
              buffer_ );
        delete[] oldBuffer;
        size_ = newSize;
    }
     string PrintSizes() {
        AINVARIANT_GUARD;
        char buf[30];
        return string("size = ") + itoa(size_,buf,10) +
               ", used = " + itoa(used_,buf,10);
    }
     bool Invariant() {
        if( used_ > 0.9*size_ ) Resize( 2*size_ );
        return used_ <= size_;
    }
private:
    char*  buffer_;
    size_t used_, size_;
};
 int f( int& x, int y = x ) { return x += y; }
int g( int& x )             { return x /= 2; }
 int main( int, char*[] ) {
```

```
    int i = 42;
    cout << "f(" << i << ") = " << f(i) << ", "
         << "g(" << i << ") = " << g(i) << endl;
    Array a(20);
    cout << a.PrintSizes() << endl;
}
```

[解答]

[“我的天哪，狮子、老虎还有狗熊！” 桃乐茜叫道。]

```
#include <cassert>
#include <iostream>
#include <typeinfo>
#include <string>
using namespace std;
//  下面的几句来自其它的头文件
char* itoa(int value, char* workArea, int radix );
extern int fileIdCounter;
```

全局变量的出现应该早已引起了我们的警觉，使我们特别留意那些企图在它被初始化之前就使用它的代码。在各翻译单元（translation units）之间的那些全局变量（包括类的静态变量）的初始化顺序并未被定义。

```
//  使类的不变量检查自动化的辅助函数
template<class T>
inline void AAssert( T& p ) {
    static int localFileId = ++fileIdCounter;
```

啊哈，这里出问题了。如果编译器恰好是在初始化任何 Aassert<T>::localFileId 之前对 fileIdCounter 进行了初始化，那么还算好。否则的话，这里的数值将会是 fileIdCounter 在被初始化之前其所占用的内存区中的内容。

```
    if( !p.Invariant() ) {
        cerr << "Invariant failed: file " << localFileId
             << ", " << typeid(p).name()
             << " at " << static_cast<void*>(&p) << endl;
        assert( false );
    }
}
template<class T>
class AInvariant {
public:
    AInvariant( T& p ) : p_(p) { AAssert( p_ ); }
    ~AInvariant()              { AAssert( p_ ); }
private:
    T& p_;
};
#define AINVARIANT_GUARD AInvariant<AIType>  \
                         invariantChecker( *this )
```

使用这些辅助性的函数是个很有意思的主意，这样可以使一个类在函数调用的前后自动的进行不变量检查。只要简单的对其来一个 AIType 的 typedef，然后再把“AINVARIANT_GUARD;”作为成员函数的第一条语句就可以了。本质上而言，这样并不完全是不好的。

然而在下面的代码中，这种做法就很不幸的变得一点都不有趣了。主要原因是 Ainvariant 隐藏了对 assert()的调用，而当程序在 non-debug 模式下被建立（build）的时候，编译器会自动的删掉 assert()。像编写下面这样的代码的程序员就很可能没有认识到这种对建立（build）模式的依赖性及其产生的相应的不同结果。

```
//-----------------------------------------------
class Array : private ArrayBase, public Container {
    typedef Array AIType;
public:
    Array( size_t startingSize = 10 )
      : Container( startingSize ),
        ArrayBase( Container::GetType() ),
```

这里的构造函数（constructor）的初始化表有两个潜在的错误。第一个错误或许不必称其为错误，但让其

留在代码里面又会形成一个扑朔迷离的遮眼法（a red herring）。我们分两点说明这个错误：

1. 如果 GetType()是一个静态成员函数，或者是一个既不使用‘this’指针（即不使用任何数据成员）又不受构造操作的副作用（比如静态的使用计数）影响的成员函数，那么这里就仅仅只是有着不良的编码风格而已，仍然能正确的运行。

2. 否则的话（一般来说是指 GetType()是普通的非静态成员函数的情况），我们就有麻烦了。非 virtual 的基类会被从左到右按照它们被声明的顺序来初始化。因此在这里，ArrayBase 会赶在 Container 之前先被初始化。不幸的是，这意味着企图使用一个尚未被初始化的 Container subobject 之成员。

```
        used_(0),
        size_(startingSize),
        buffer_(new char[size_])
```

这第二个错误是无庸置疑的，因为实际上变量会以它们在类定义里面出现的顺序被初始化：

```
        buffer_(new char[size_])
        used_(0),
        size_(startingSize),
```

像这样写，错误就很明显了。对 new[]的调用会使 buffer 得到一个无法预测的大小——一般为 0 或者很大，这取决于编译器在调用构造函数（constructor）之前是否恰好将对象的内存区初始化为空（null）。无论如何，初始化分配的空间大小都几乎不可能为 startingSize。

```
    {
        AINVARIANT_GUARD;
    }
```

效率方面的小问题：在这里，Invariant()函数没有必要被调用两次
（分别是在潜在的临时对象的构造过程和析构过程中）。当然这只是小问题，
不会引起大麻烦。

```
    void Resize( size_t newSize ) {
        AINVARIANT_GUARD;
        char* oldBuffer = buffer_;
        buffer_ = new char[newSize];
        memset( buffer_, ' ', newSize );
        copy( oldBuffer, oldBuffer+min(size_,newSize),
              buffer_ );
        delete[] oldBuffer;
        size_ = newSize;
    }
```

这里存在一个严重的控制流（control flow）方面的问题。我从未见人指出过这一点（如果你曾指出过的话，那么我真的很抱歉），而其实这是我故意设计的，就是想看看有没有人会指出来。

在叙述这一点之前，你可以再读一遍代码，看看自己到底能不能发现这个问题（提示：其实是显而易见的）。

* * * * * * * * * * * * * *

好了，答案即：代码不是异常-安全的（exception-safe）。如果对 new[]的调用导致抛出一个异常的话，那么不但当前的对象会处在一个无效的状态，而且原来的 buffer 还会出现内存泄漏的情况，因为所有指向它的指针都丢失了从而导致不能将其删除掉。

这个函数的问题说明了，迄今为止，几乎还没有程序员养成了编写异常-安全的（excepton-safe）代码的习惯——即使是在前不久的 GotW 条款中对异常安全性（exception safety）作了广泛的讨论之后！

```
    string PrintSizes() {
        AINVARIANT_GUARD;
        char buf[30];
        return string("size = ") + itoa(size_,buf,10) +
               ", used = " + itoa(used_,buf,10);
    }
```

其中，itoa()原型函数使用 buf 作为存放结果的地方（译注：意即这一次调用该函数时会覆盖上一次调用该函数时所设置的 buf 的内容，从而引起了潜在的顺序问题）。这段代码里也存在着控制流（control flow）方面的问题。我们无法预计最后那个返回语句中对表达式的求值顺序，因为对函数参数的操作顺序是没有明确规定的，其完全取决于特定的实现方案。（这里还要注意，内建（built-in）的 operator+不存在这种问题，然而一旦你提供了自己的 operator+版本，那么你的版本就会视为函数调用。）

最后那个返回语句所表现出来的问题的严重性在下面这个语句中得到了更好的展示（就算对 operator+的调用是从左到右的）：

```
return
 operator+(
   operator+(
     operator+( string("size = "),
                itoa(size_,buf,10) ) ,
     ", used = " ) ,  //译注：这里原文是", y = " ) ,应该是写错了
   itoa(used_,buf,10) );
```

这里我们假设 size_值为 10，used_值为 5。如果最外面的 operator+()的第一个参数先被求值的话，那么结果将会是正确的“size = 10，used = 5”，因为第一个 itoa()函数存放在 buf 里的结果会在第二个 itoa()函数复用 buf 之前就被读出使用。但如果最外面的 operator+()的第二个参数先被求值的话（例如在 MSVC 4.X 上就是这样），那么结果将会是错误的“size = 10，used = 10”，因为外层的那个 itoa()函数先被求值，但其结果（译注：即字符‘5’）会在被使用之前就又被内层的那个 itoa()函数毁掉了（译注：buf 里面变成了‘10’）。

```
    bool Invariant() {
        if( used_ > 0.9*size_ ) Resize( 2*size_ );
        return used_ <= size_;
```

对 Resize()的调用存在两个问题：

1. 这种情况下，程序压根儿就不会正常工作，因为如果条件判断为真，那么 Resize()会被调用，这又会导致立即再次调用 Invarient()；接着条件判断仍然会为真，然后再调用 Resize()，这又会导致立即再次调用 Invarient()；接着……你一定明白这是什么问题了（译注：这是一个无法终止的递归调用）。
2. 如果 AAssert()的编写者出于效率方面的考虑把错误报告（error reporting）的代码删掉并取而代之以“assert(p->Invarient());”，那又会如何呢？其结果只会使这里的代码变得更可悲，因为其在 assert()调用中加入了会产生副作用的代码。这意味着程序在 debug mode 和 release mode 两种不同的编译模式下产生的可执行代码在执行时会具有不同的行为。即使没有上面第 1 点中说明的问题，这也是很不好的，因为这就意味着 Array 对象会依据建立模式（build mode）的不同而 Resize 不同的次数，并使软件测试人员从此过上人间地狱般的生活——当他试图在一个以 debug mode 建立的程序里重现客户遇到的问题（译注：当然，客户拿到的程序是以 release mode 建立并发布的）时，他得到的运行期内存映像之特征与 release mode 下的运行期内存映像之特征是不一样的。

[概要]：绝不要在对 assert()的掉用中加入有副作用的代码，并且总是确认递归过程肯定会终止。

```
    }
private:
    char*  buffer_;
    size_t used_, size_;
};
int f( int& x, int y = x ) { return x += y; }
```

那第二个设置缺省值的参数无论如何都不能算是一个合法的 C++用法，在一个理想的编译器下应该编译不通过（尽管现实中有些编译器可以将其编译通过）。说这个用法不好还是因为编译器可以采用任意的顺序来对函数参数求值，y 可能赶在 x 之前先被初始化。

```
int g( int& x )            { return x /= 2; }
int main( int, char*[] ) {
    int i = 42;
    cout << "f(" << i << ") = " << f(i) << ", "
         << "g(" << i << ") = " << g(i) << endl;
```

这里还是对参数求值的顺序问题。由于没有确定 f(i)和 g(i)被求值的先后顺序（甚或是两个对 i 本身的求值顺序），因此显示出来的结果可能是错误的。MSVC 中的结果是“f(22)=22,g(21)=21”；这也就是说，其编译器按照从右往左的顺序对函数参数进行求值。

但是，你会说，这个结果不是错的吗？告诉你吧，结果没错（!），编译器也没出错……而且，另外一个编译器可能会出现完全不同的结果，你也还是不能归咎于编译器，因为其结果依赖于一个在 C++里没有被明确定义的操作过程。

```
    Array a(20);
    cout << a.PrintSizes() << endl;
}
```

也许桃乐茜在本条款开头说的话并不完全正确……下面这句话大概更接近真相：

[“我的天哪，参数、全局变量还有异常处理！” 桃乐茜在进修了中级 C++课程之后叫道。]

12. 後记

如果你欣赏本书所列的难题和讨论，那么我有好消息要告诉你。Guru of the Week#30 并不是一个结束，它并不是最後一篇 GotW，我也并非不再为电脑杂志撰写文章。

如今，新的 GotW 已经常态性地在网际网路讨论群组 comp.lang.c++.moderated 上发表，所有档案并收集於 GotW 官方网站 www.PeerDirect.com/resources。在我下笔此刻，也就是 1999 年六月，我们已经进行到#55 篇了。为了让你体验一下即将来临的有些什么，以下是 GotW 刊物的部份取样：

- 流行题目上的更多探讨，包括如何安全使用 auto_ptr 和 namespaces、exception-safety 的主题和技术、条款 8~17,31~34,37 的下一步。
- 系列文章，介绍 reference-counting 和 copy-on-write 技术，包括多绪（或有多绪能力的）环境中的效率关联，搭配广泛的测试和统计学上的量测。这些都是你在别的地方不易见到的菜色。
- 如何安全而有效地运用标准程式库，特别是容器（如 vector, map）和标准 streams。其中也包括如何在条款 2 和 3 的精神下以最佳方式扩充标准程式库。
- 一个极好的游戏：以最少的指令撰写一个 MasterMind-playing 程式。

这只是少量取样。如果你手上这本书对你而言有足够的趣味，我打算再写另一本，将我为 C++ Report, C/C++ Users Journal 及其他杂志撰写的 C++ 工程文章扩充并重新组织後付梓。

希望你从本书获得乐趣，也希望你继续让我知道未来你对什么感兴趣、想看些什么；请进入前述网站，那儿会告诉你如何提交你的需求。本书的某些题目就是以那样的电子邮件形式发起的。

再次感谢所有表达兴趣并支持 GotW 和本书的人。我希望你从中找到对你日常工作有帮助的东西，让你能够撰写更快、更清爽、更安全的 C++ 程式。

本书内容源自极受欢迎的网际网路论坛 C++ 特别节目 Guru of the Week

Exceptional C++ 以实例进行的方式，告诉你如何以标准 C++ 进行软体工程。你乐於解决棘手的 C++ 疑难杂症吗？你嗜好写出稳健强固而且具有扩充性的程式码吗？那么，请花数分钟时间，拿书中一些强韧的 C++ 设计问题和实作问题来挑战自己。

Exceptional C++ 书中的疑难杂症并非只具有娱乐性质而已，它们可以帮助你磨练你的技巧，使你成为最顶尖的 C++ 程式设计者。其中许多问题精选自网际网路讨论群 comp.lang.c++.moderated 上著名的 Guru of the Week 特别节目，并加以扩充、修改，以符合正式的 ISO/ANSI C++ 标准规格。

每个问题都有一个难度评分，每个问题都用来说明敏感、绝妙、不可思议的一些程式设计上的错误或考量。你可以先试著自行解决问题，然後本书会剖析程式码，说明错误在哪里，告诉你如何解决问题。这些问题涵盖广泛，包括以下几个重要主题：

! 泛型程式设计；如何撰写可重用的 templates
! 异常情况（exceptions）发生时仍然安全的写码技术
! 强固稳健的 class 设计与 class 继承
! 编译器防火墙与所谓的 Pimpl Idiom
! 名称搜寻（Name Lookup）、命名空间（namespaces）及介面原则
! 记忆体管理相关主题与技术
! 陷阱、易犯错误与反常作法
! 最佳化

试著跟随 C++ 大师磨练你的技巧，并以本书的深刻洞察和丰富经验，产生更有效率、更具效益、更稳健强固、更容易移植的 C++ 码。

Herb Sutter，Guru of the Week 专栏的开创者，PeerDirect Inc. 的首席技术员，PeerDirect 异型资料库产品的主要架构者。他把最近 12 年的时间花在资料库和通讯谘询领域，主要以 C++ 完成工作。他拥有数项专利，范围涵盖分散式资料库、分散式网路、资料库密码学等数十项创新发明。他是 ISO/ANSI C++ 和 SOL 标准委员会的选举委员，C++ Report 的专栏作家，同时也是网际网路讨论群 comp.lang.c++.moderated 的创始主持人。

侯捷 集电脑技术性读物之著、译、评於一身，在台湾软体界广受尊敬。他是《多型与虚拟》和《泛型技术》的作者，《C++ Primer 中文版》等书的译者，Run!PC 杂志的知名专栏作家，也是极具号召力的大学教师与研讨会讲师。