More Effecitve C++

作者	:	Scott Meyers
译序、导读	:	侯捷
Item 1～28	:	ZHC
Item 29～35	:	WQ
附 1	:	侯捷
附 2	:	WQ
附 3、附 4	:	陈崴

1. 译序（侯捷）

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（Design Patterns,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。本书不但与原文本页页对译，保留索引，并加上精装、书签条、译注、书籍交叉参考 1、完整范例码 2、读者服务 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 2000/05/15 于新竹.台湾
jjhou@ccca.nctu.edu.tw
http://www.jjhou.com

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。

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。

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）。

2. 导读

对 C++ 程式员而言，日子似乎有点过於急促。虽然只商业化不到 10 年，C++ 却俨然成为几乎所有主要电算环境的系统程式语言霸主。面临程式设计方面极具挑战性问题的公司和个人，不断投入 C++ 的怀抱。而那些尚未使用 C++ 的人，最常被询问的一个问题则是：你打算什么时候开始用 C++。C++ 标准化已经完成，其所附带之标准程式库幅员广大，不仅涵盖 C 函式库，也使之相形见绌。这么一个大型程式库使我们有可能在不必牺牲移植性的情况下，或是在不必从头撰写常用演算法和资料结构的情况下，完成琳琅满目的各种复杂程式。C++ 编译器的数量不断增加，它们所供应的语言性质不断扩充，它们所产生的码品质也不断改善。C++ 开发工具和开发环境愈来愈丰富，威力愈来愈强大，稳健强固（robust）的程度愈来愈高。商业化程式库几乎能够满足各个应用领域中的写码需求。

一旦语言进入成熟期，而我们对它的使用经验也愈来愈多，我们所需要的资讯也就随之改变。1990 年人们想知道 C++ 是什么东西。到了 1992 年，他们想知道如何运用它。如今 C++ 程式员问的问题更高级：我如何能够设计出适应未来需求的软体？我如何能够改善程式码的效率而不折损正确性和易用性？我如何能够实作出语言未能直接支援的精巧机能？

这本书中我要回答这些问题，以及其他许多类似问题。

本书告诉你如何更具实效地设计并实作 C++ 软体：让它行为更正确；面对异常情况时更稳健强固；更有效率；更具移植性；将语言特性发挥得更好；更优雅地调整适应；在「混合语言」开发环境中运作更好；更容易被正确运用；更不容易被误用。简单地说就是如何让软体更好。

本书内容分为 35 个条款。每个条款都在特定主题上精简摘要出 C++ 程式设计社群所累积的智慧。大部份条款以准则的型式呈现，附随的说明则阐述这条准则为什么存在，如果不遵循会发生什么後果，以及什么情况下可以合理违反该准则。所有条款被我分为数大类。某些条款关心特定的语言性质，特别是你可能罕有使用经验的一些新性质。例如条款 9~15 专注於 exceptions（就像 Tom Cargill, Jack Reeves, Herb Sutter 所发表的那些杂志文章一样）。其他条款解释如何结合语言的不同特性以达成更高阶目标。例如条款 25~31 描述如何限制物件的个数或诞生地点，如何根据一个以上的物件型别产生出类似虚拟函式的东西，如何产生 smart pointers 等等。其他条款解决更广泛的题目。条款 16~24 专注於效率上的议题。不论哪一条款，提供的都是与其主题相关且意义重大的作法。在 More Effective C++ 一书中你将学习到如何更实效更精锐地使用 C++。大部份 C++ 教科书中对语言性质的大量描述，只能算是本书的一个背景资讯而已。

这种处理方式意味，你应该在阅读本书之前便熟悉 C++。我假设你已了解类别（classes）、保护层级（protection levels）、虚拟函式、非虚拟函式，我也假设你已通晓 templates 和 exceptions 背後的概念。我并不期望你是一位语言专家，所以涉及较罕见的 C++ 特性时，我会进一步做解释。

2.1 本书所谈的 C++

我在本书所谈、所用的 C++，是 ISO/ANSI 标准委员会於 1997 年 11 月完成的 C++国际标准最後草案（Final Draft International Standard）。这暗示了我所使用的某些语言特性可能并不在你的编译器(s) 支援能力之列。别担心，我认为对你而言唯一所谓「新」特性，应该只有 templates，而 templates 如今几乎已是各家编译器的必备机能。我也运用 exceptions，并大量集中於条款 9~15。如果你的编译器(s) 未能支援 exceptions，没什么大不了，这并不影响本书其他部份带给你的好处。但是，听我说，纵使你不需用到 exceptions，亦应阅读条款 9~15，因为那些条款（及其相关篇幅）检验了某些不论什么场合下你都应该了解的主题。

我承认，就算标准委员会授意某一语言特性或是赞同某一实务作法，并非就保证该语言特性已出现在目前的编译器上，或该实务作法已可应用於既有的开发环境上。一旦面对「标准委员会所议之理论」和「真正能够有效运作之实务」间的矛盾，我便两者都加以讨论，虽然我其实比较更重视实务。由於两者我都讨论，所以当你的编译器(s) 和 C++ 标准不一致时，本书可以协助你，告诉你如何使用目前既有的架构来模拟编译器(s) 尚未支援的语言特

性。而当你决定将一些原本绕道而行的解决办法以新支援的语言特性取代时，本书亦可引导你。

注意当我说到编译器(s) 时，我使用复数。不同的编译器对 C++ 标准的满足程度各不相同，所以我鼓励你在至少两种编译器(s) 平台上发展程式码。这么做可以帮助你避免不经意地依赖某个编译器专属的语言延伸性质，或是误用某个编译器对标准规格的错误阐示。这也可以帮助你避免使用过度先进的编译器技术，例如独家厂商才做得出的某种语言新特性。如此特性往往实作不够精良（臭虫多，要不就是表现迟缓，或是两者兼具），而且 C++ 社群往往对这些特性缺乏使用经验，无法给你应用上的忠告。雷霆万钧之势固然令人兴奋，但当你的目标是要产出可靠的码，恐怕还是步步为营（并且能够与人合作）得好。

本书用了两个你可能不甚熟悉的 C++ 性质，它们都是晚近才加入 C++ 标准之中。某些编译器支援它们，但如果你的编译器不支援，你可轻易以你所熟悉的其他性质来模拟它们。

第一个性质是型别 bool，其值必为关键字 true 或 false。如果你的编译器尚未支援 bool，有两个方法可以模拟它。第一个方法是使用一个 global enum：enum bool { false, true };
这允许你将参数为 bool 或 int 的不同函式加以多载化（overloading）。缺点是，内建的「比较运算子（comparison operators）」如==, <, >=, 等等仍旧传回 ints。

所以以下程式码的行为不如我们所预期：

```
void f(int);
void f(bool);
int x, y;
...
f( x < y ); // 呼叫 f(int)，但其实它应该呼叫 f(bool)
```

一旦你改用真正支援 bool 的编译器，这种 enum 近似法可能会造成程式行为的改变。

另一种作法是利用 typedef 来定义 bool，并以常数物件做为 true 和 false：

```
typedef int bool;
const bool false = 0;
const bool true = 1;
```

这种手法相容於传统的 C/C++ 语意。使用这种模拟法的程式，在移植到一个支援有 bool 型别的编译器平台之後，行为并不会改变。缺点则是无法在函式多载化（overloading）时区分 bool 和 int。以上两种近似法都有道理，请选择最适合你的一种。

第二个新性质，其实是四个转型运算子：static_cast, const_cast, dynamic_cast, 和 reinterpret_cast。如果你不熟悉这些转型运算子，请翻到条款 2 仔细阅读其中内容。它们不只比它们所取代的 C 旧式转型做得更多，也更好。书中任何时候当我需要执行转型动作，我都使用新式的转型运算子。

C++ 拥有比语言本身更丰富的东西。是的，C++ 还有一个伟大的标准程式库（见条款 E49）。我尽可能使用标准程式库所提供的 string 型别来取代 char* 指标，而且我也鼓励你这么做。string objects 并不比 char*-based 字串难操作，它们的好处是可以免除你大部份的记忆体管理工作。而且如果发生 exception 的话（见条款 9 和 10），string objects 比较没有 memory leaks（记忆体遗失）的问题。

实作良好的 string 型别甚至可和对应的 char* 比赛效率，而且可能会赢（条款 29 会告诉你个中故事）。如果你不打算使用标准的 string 型别，你当然会使用类似 string 的其他 classes，是吧？是的，用它，因为任何东西都比直接使用 char* 来得好。

我将尽可能使用标准程式库提供的资料结构。这些资料结构来自 Standard Template Library（"STL" — 见条款 35）。STL 包含 bitsets, vectors, lists, queues, stacks, maps, sets, 以及更多东西，你应该尽量使用这些标准化的资料结构，不要情不自禁地想写一个自己的版本。你的编译器或许没有附 STL 给你，但不要因为这样就不使用它。感谢 Silicon Graphics 公司的热心，你可以从 SGI STL 网站下载一份免费产品，它可以和多种编译器搭配。

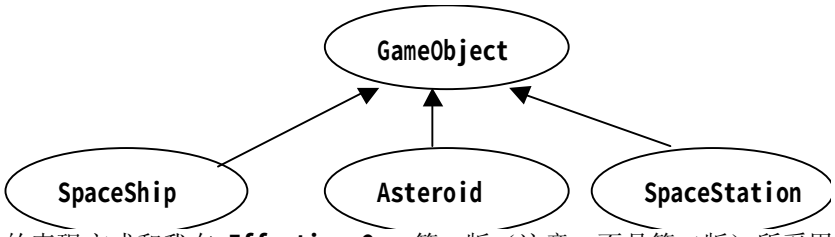
如果你目前正在使用一个内含各种演算法和资料结构的程式库，而且用得相当愉快，那么就没有必要只为了「标准」两个字而改用 STL。然而如果你在「使用 STL」和「自行撰写

同等功能的码」之间可以选择，你应该让自己倾向使用 **STL**。记得程式码的重用性吗？**STL**（以及标准程式库的其他组件）之中有许多码是十分值得重复运用的。

2.2 惯例与术语

任何时候如果我谈到 **inheritance**（继承），我的意思是 **public inheritance**（见条款 **E35**）。如果我不是指 **public inheritance**，我会明白地指明。绘制继承体系图时，我对 **base-derived** 关系的描述方式，是从 **derived classes** 往 **base classes** 画箭头。

例如，下面是条款 **31** 的一张继承体系图：



这样的表现方式和我在 **Effective C++** 第一版（注意，不是第二版）所采用的习惯不同。现在我决定使用这种最广被接受的继承箭头画法：从 **derived classes** 画往 **base classes**，而且我很高兴事情终能归於一统。此类示意图中，抽象类别（**abstract classes**，例如上图的 **GameObject**）被我加上阴影而具象类别（**concrete classes**，例如上图的 **SpaceShip**）未加阴影。

Inheritance（继承机制）会引发「**pointers**（或 **references**）拥有两个不同型别」的议题，两个型别分别是静态型别（**static type**）和动态型别（**dynamic type**）。**Pointer** 或 **reference** 的「静态型别」是指其宣告时的型别，「动态型别」则由它们实际所指的物件来决定。下面是根据上图所写的一个例子：

```
GameObject *pgo = // pgo 的静态型别是 GameObject*，
new SpaceShip; // 动态型别是 SpaceShip*
Asteroid *pa = new Asteroid; // pa 的静态型别是 Asteroid*，
// 动态型别也是 Asteroid*。
pgo = pa; // pgo 的静态型别仍然（永远）是 GameObject*，
// 至於其动态型别如今是 Asteroid*。
GameObject& rgo = *pa; // rgo 的静态型别是 GameObject，
// 动态型别是 Asteroid。
```

这些例子也示范了我喜欢的一种命名方式。**pgo** 是一个 **pointer-to-GameObject**；**pa** 是一个 **pointer-to-Asteroid**；**rgo** 是一个 **reference-to-GameObject**。我常常以此方式来为 **pointer** 和 **reference** 命名。

我很喜欢两个参数名称：**lhs** 和 **rhs**，它们分别是**"left-hand side"** 和**"right-hand side"** 的缩写。为了了解这些名称背後的基本原理，请考虑一个用来表示分数（**rational numbers**）的 **class**：

```
class Rational { ... };
```

如果我想要一个用以比较两个 **Rational objects** 的函式，我可能会这样宣告：

```
bool operator==(const Rational& lhs, const Rational& rhs);
```

这使我得以写出这样的码：

```
Rational r1, r2;
...
if (r1 == r2) ...
```

在呼叫 **operator==** 的过程中，**r1** 位於**"=="** 左侧，被系结於 **lhs**，**r2** 位於**"=="**右侧，被系结於 **rhs**。

我使用的其他缩写名称还包括：**ctor** 代表**"constructor"**，**dtor** 代表**"destructor"**，**RTTI** 代表 **C++** 对 **runtime type identification** 的支援（在此性质中，**dynamic_cast** 是最常被使用的一个零组件）。

当你配置记忆体而没有释放它，你就有了 **memory leak**（记忆体遗失）问题。**Memory leaks** 在 **C** 和 **C++** 中都有，但是在 **C++** 中，**memory leaks** 所遗失的还不只是记忆体，因为 **C++** 会在物件被产生时，自动呼叫 **constructors**，而 **constructors** 本身可能亦配有资源（**resources**）。举个例子，考虑以下程式码：

```
class Widget { ... }; // 某个 class — 它是什么并不重要。
Widget *pw = new Widget; // 动态配置一个 Widget 物件。
... // 假设 pw 一直未被删除（deleted）。
```

这段码会遗失记忆体，因为 **pw** 所指的 **Widget** 物件从未被删除。如果 **Widget constructor** 配置了其他资源（例如 **file descriptors, semaphores, window handles,database locks**），这些资源原本应该在 **Widget** 物件被摧毁时释放，现在也像记忆体一样都遗失掉了。为了强调在 **C++** 中 **memory leaks** 往往也会遗失其他资源，我在书中常以 **resource leaks** 一词取代 **memory leaks**。

你不会在本书中看到许多 **inline** 函式。并不是我不喜欢 **inlining**，事实上我相信 **inline** 函式是 **C++** 的一项重要性质。然而决定一个函式是否应被 **inlined**，条件十分复杂、敏感、而且与平台有关（见条款 **E33**）。所以我尽量避免 **inlining**，除非其中有个关键点非使用 **inlining** 不可。当你在本书之中看到一个 **non-inline** 函式，并不意味我认为把它宣告为 **inline** 是个坏主意，而只是说，它「是否为 **inline**」与当时讨论的主题无关。

有一些传统的 **C++** 性质已明白地被标准委员会排除。这样的性质被明列於语言的最後撤除名单，因为新性质已经加入，取代那些传统性质的原本工作，而且做得更好。这本书中我会检视被撤除的性质，并说明其取代者。你应该避免使用被撤除的性质，但是过度在意倒亦不必，因为编译器厂商为了挽留其客户，会尽力保存回溯相容性，所以那些被撤除的性质大约还会存活好多年。

所谓 **client**，是指你所写的程式码的客户。或许是某些人（程式员），或许是某些物（**classes** 或 **functions**）。举个例子，如果你写了一个 **Date class**（用来表现生日、最後期限、耶稣再次降临日等等），任何使用了这个 **class** 的人，便是你的 **client**。任何一段使用了 **Date class** 的码，也是你的 **clients**。**Clients** 是重要的。

事实上 **clients** 是游戏的主角。如果没有人使用你写的软体，你又何必写它呢？你会发现我很在意如何让 **clients** 更轻松，通常这会导至你的行事更困难，因为好的软体「以客为尊」。如果你讥笑我太过滥情，不妨反躬自省一下。你曾经使用过自己写的 **classes** 或 **functions** 吗？如果是，你就是你自己的 **client**，所以让 **clients** 更轻松，其实就是让自己更轻松，利人利己。

当我讨论 **class template** 或 **function templates** 以及由它们所产生出来的 **classes** 或 **functions** 时，请容我保留偷懒的权利，不一一写出 **templates** 和其 **instantiations**（具现体）之间的差异。举个例子，如果 **Array** 是个 **class template**，有个型别参数 **T**，我可能会以 **Array** 代表此 **template** 的某个特定具现体（**instantiation**），虽然其实 **Array<T>** 才是正式的 **class** 名称。同样道理，如果 **swap** 是个 **function template**，有个型别参数 **T**，我可能会以 **swap** 而非 **swap<T>** 表示其具现体。如果这样的简短表示法在当时情况下不够清楚，我便会在表示 **template** 具现体时加上 **template** 参数。

2.3 臭虫报告，意见提供，内容更新

我尽力让这本书技术精准、可读性高，而且有用，但是我知道一定仍有改善空间。

如果你发现任何错误— 技术性的、语言上的、错别字、或任何其他东西— 请告诉我。我会试著在本书新刷中修正之。如果你是第一位告诉我的人，我会很高兴将你的大名登录到本书致谢文（**acknowledgments**）内。如果你有改善建议，我也非常欢迎。

我将继续收集 **C++** 程式设计的实效准则。如果你有任何这方面的想法并愿意与我分享，我会十分高兴。请将你的建议、你的见解、你的批评、以及你的臭虫报告，寄至：

Scott Meyers
c/o Editor-in-Chief, Corporate and Professional Publishing
Addison-Wesley Publishing Company
1 Jacob Way

Reading, MA 01867
U. S. A.

或者你也可以送电子邮件到 mec++@awl.com。

我维护有一份本书第一刷以来的修订记录，其中包括错误修正、文字修润、以及技术更新。你可以从本书网站取得这份记录，以及其他与本书相关的资讯。你也可以透过 anonymous FTP，从 ftp.awl.com 的 cp/mec++ 目录中取得它。如果你希望拥有这份资料，但无法上网，请寄申请函到上述地址，我会邮寄一份给你。

这篇序文有够长的，让我们开始正题吧。

3. 基础议题

基础议题。是的，**pointers**（指针）、**references**（引用）、**casts**（类型转换）、**arrays**（数组）、**constructors**（构造）- 再没有比这些更基础的议题了。几乎最简单的 C++ 程序也会用到其中大部份特性，而许多程序会用到上述所有特性。

尽管你可能已经十分熟悉语言的这一部份，有时候它们还是会令你吃惊。特别是对那些从 C 转到 C++ 的程序员，因为 **references, dynamic casts, default constructors** 及其它 non-C 性质背后的观念，往往带有一股幽暗阴郁的色彩。

這一章描述 **pointers** 和 **references** 的差异，並告诉你它们的适当使用时机。本章介绍新的 C++ 类型转换（**casts**）语法，並解释为什么新式类型转换法比旧式的 C 类型转换法优越。本章也检验 C 的数组概念以及 C++ 的多态（**polymorphism**）概念，並说明为什么將这两者混合运用是不智之举。最后，本章讨论 **default constructors**（默认构造函数）的正方和反方意见，並提出一些建议作法，让你回避语言的束缚（因为在你不需 **default constructors** 的情况下，C++ 也会给你一个。

只要留心下面各条款的各项忠告，你将向著一個很好的目标迈进：你所生产的软件可以清楚而正确地表現出你的设计意图。

3.1 Item M1：指针与引用的区别

指针与引用看上去完全不同（指针用操作符“*”和“->”，引用使用操作符“.”），但是它们似乎有相同的功能。指针与引用都是让你间接引用其他对象。你如何决定在什么时候使用指针，在什么时候使用引用呢？

首先，要认识到在任何情况下都不能使用指向空值的引用。一个引用必须总是指向某些对象。因此如果你使用一个变量并让它指向一个对象，但是该变量在某些时候也可能不指向任何对象，这时你应该把变量声明为指针，因为这样你可以赋空值给该变量。相反，如果变量肯定指向一个对象，例如你的设计不允许变量为空，这时你就可以把变量声明为引用。

“但是，请等一下”，你怀疑地问，“这样的代码会产生什么样的后果？”

```
char *pc = 0;           // 设置指针为空值
char& rc = *pc;         // 让引用指向空值
```

这是非常有害的，毫无疑问。结果将是不确定的（编译器能产生一些输出，导致任何事情都有可能发生）。应该躲开写出这样代码的人，除非他们同意改正错误。如果你担心这样的代码会出现在你的软件里，那么你最好完全避免使用引用，要不然就去让更优秀的程序员去做。我们以后将忽略一个引用指向空值的可能性。

因为引用肯定会指向一个对象，在 C＋＋里，引用应被初始化。

```
string& rs;             // 错误，引用必须被初始化
string s("xyzzy");
string& rs = s;         // 正确，rs 指向 s
```

指针没有这样的限制。

```
string *ps;             // 未初始化的指针
                        // 合法但危险
```

不存在指向空值的引用这个事实意味着使用引用的代码效率比使用指针的要高。因为在使用引用之前不需要测试它的合法性。

```
void printDouble(const double& rd)
{
    cout << rd;         // 不需要测试 rd,它
}                       // 肯定指向一个 double 值
```

相反，指针则应该总是被测试，防止其为空：

```
void printDouble(const double *pd)
{
  if (pd) {             // 检查是否为 NULL
    cout << *pd;
 }
}
```

指针与引用的另一个重要的不同是指针可以被重新赋值以指向另一个不同的对象。但是引用则总是指向在初始化时被指定的对象，以后不能改变。

```
string s1("Nancy");
string s2("Clancy");
string& rs = s1;          // rs 引用 s1
string *ps = &s1;         // ps 指向 s1
rs = s2;                 // rs 仍旧引用 s1,
                        // 但是 s1 的值现在是
                        // "Clancy"
ps = &s2;                // ps 现在指向 s2;
                        // s1 没有改变
```

总的来说，在以下情况下你应该使用指针，一是你考虑到存在不指向任何对象的可能(在这种情况下，你能够设置指针为空)，二是你需要能够在不同的时刻指向不同的对象（在这种情况下，你能改变指针的指向)。如果总是指向一个对象并且一旦指向一个对象后就不会改变指向，那么你应该使用引用。

还有一种情况，就是当你重载某个操作符时，你应该使用引用。最普通的例子是操作符[]。这个操作符典型的用法是返回一个目标对象，其能被赋值。

```
vector<int> v(10);        // 建立整形向量（vector），大小为10;
                          // 向量是一个在标准C库中的一个模板(见条款M35)
v[5] = 10;                // 这个被赋值的目标对象就是操作符[]返回的值
```

如果操作符[]返回一个指针，那么后一个语句就得这样写：

```
*v[5] = 10;
```

但是这样会使得 v 看上去象是一个向量指针。因此你会选择让操作符返回一个引用。（这有一个有趣的例外，参见条款M30）

当你知道你必须指向一个对象并且不想改变其指向时，或者在重载操作符并为防止不必要的语义误解时，你不应该使用指针。而在除此之外的其他情况下，则应使用指针。

3.2 Item M2：尽量使用C++风格的类型转换

仔细想想地位卑贱的类型转换功能（cast），其在程序设计中的地位就象goto语句一样令人鄙视。但是它还不是无法令人忍受，因为当在某些紧要的关头，类型转换还是必需的，这时它是一个必需品。

不过C风格的类型转换并不代表所有的类型转换功能。

一来它们过于粗鲁，能允许你在任何类型之间进行转换。不过如果要进行更精确的类型转换，这会是一个优点。在这些类型转换中存在着巨大的不同，例如把一个指向const对象的指针（pointer-to-const-object）转换成指向非const对象的指针（pointer-to-non-const-object）(即一个仅仅去除const的类型转换)，把一个指向基类的指针转换成指向子类的指针（即完全改变对象类型）。传统的C风格的类型转换不对上述两种转换进行区分。(这一点也不令人惊讶，因为C风格的类型转换是为C语言设计的，而不是为C++语言设计的)。

二来C风格的类型转换在程序语句中难以识别。在语法上，类型转换由圆括号和标识符组成，而这些可以用在C＋＋中的任何地方。这使得回答象这样一个最基本的有关类型转换的问题变得很困难：“在这个程序中是否使用了类型转换？”。这是因为人工阅读很可能忽略了类型转换的语句，而利用象grep的工具程序也不能从语句构成上区分出它们来。

C++通过引进四个新的类型转换操作符克服了C风格类型转换的缺点，这四个操作符是，static_cast，const_cast，dynamic_cast，和reinterpret_cast。在大多数情况下，对于这些操作符你只需要知道原来你习惯于这样写，

```
(type) expression
```

而现在你总应该这样写：

```
static_cast<type>(expression)
```

例如，假设你想把一个 **int** 转换成 **double**，以便让包含 **int** 类型变量的表达式产生出浮点数值的结果。如果用 **C** 风格的类型转换，你能这样写：

```
int firstNumber, secondNumber;
...
double result = ((double)firstNumber)/secondNumber;
```

如果用上述新的类型转换方法，你应该这样写：

```
double result = static_cast<double>(firstNumber)/secondNumber;
```

这样的类型转换不论是对人工还是对程序都很容易识别。

static_cast 在功能上基本上与 **C** 风格的类型转换一样强大，含义也一样。它也有功能上限制。例如，你不能用 **static_cast** 象用 **C** 风格的类型转换一样把 **struct** 转换成 **int** 类型或者把 **double** 类型转换成指针类型，另外，**static_cast** 不能从表达式中去除 **const** 属性，因为另一个新的类型转换操作符 **const_cast** 有这样的功能。

其它新的 **C++**类型转换操作符被用在需要更多限制的地方。**const_cast** 用于类型转换掉表达式的 **const** 或 **volatileness** 属性。通过使用 **const_cast**，你向人们和编译器强调你通过类型转换想做的只是改变一些东西的 **constness** 或者 **volatileness** 属性。这个含义被编译器所约束。如果你试图使用 **const_cast** 来完成修改 **constness** 或者 **volatileness** 属性之外的事情，你的类型转换将被拒绝。下面是一些例子：

```
class Widget { ... };
class SpecialWidget: public Widget { ... };
void update(SpecialWidget *psw);
SpecialWidget sw;                    // sw 是一个非 const 对象。
const SpecialWidget& csw = sw;    // csw 是 sw 的一个引用
                                     // 它是一个 const 对象
update(&csw);  // 错误!不能传递一个 const SpecialWidget* 变量
               // 给一个处理 SpecialWidget*类型变量的函数
update(const_cast<SpecialWidget*>(&csw));
            // 正确，csw 的 const 被显示地转换掉（
            // csw 和 sw 两个变量值在 update
            //函数中能被更新）
update((SpecialWidget*)&csw);
                        // 同上，但用了一个更难识别
                        //的 C 风格的类型转换
Widget *pw = new SpecialWidget;
update(pw);          // 错误！pw 的类型是 Widget*，但是
```

```
                    // update 函数处理的是 SpecialWidget*类型
update(const_cast<SpecialWidget*>(pw));
                    // 错误！const_cast 仅能被用在影响
                    // constness or volatileness 的地方上。,
                    // 不能用在向继承子类进行类型转换。
```

到目前为止，const_cast 最普通的用途就是转换掉对象的 const 属性。

第二种特殊的类型转换符是 dynamic_cast，它被用于安全地沿着类的继承关系向下进行类型转换。这就是说，你能用 dynamic_cast 把指向基类的指针或引用转换成指向其派生类或其兄弟类的指针或引用，而且你能知道转换是否成功。失败的转换将返回空指针（当对指针进行类型转换时）或者抛出异常（当对引用进行类型转换时）：

```
Widget *pw;
...
update(dynamic_cast<SpecialWidget*>(pw));
          // 正确，传递给 update 函数一个指针
          // 是指向变量类型为 SpecialWidget 的 pw 的指针
          // 如果 pw 确实指向一个对象,
          // 否则传递过去的将使空指针。
void updateViaRef(SpecialWidget& rsw);
updateViaRef(dynamic_cast<SpecialWidget&>(*pw));
                    //正确。 传递给 updateViaRef 函数
                    // SpecialWidget pw 指针，如果 pw
                    // 确实指向了某个对象
                    // 否则将抛出异常
```

dynamic_casts 在帮助你浏览继承层次上是有限制的。它不能被用于缺乏虚函数的类型上（参见条款 M24），也不能用它来转换掉 constness：

```
int firstNumber, secondNumber;
...
double result = dynamic_cast<double>(firstNumber)/secondNumber;
                    // 错误！没有继承关系
const SpecialWidget sw;
...
update(dynamic_cast<SpecialWidget*>(&sw));
                    // 错误! dynamic_cast 不能转换
                    // 掉 const。
```

如你想在没有继承关系的类型中进行转换，你可能想到 **static_cast**。如果是为了去除 **const**，你总得用 **const_cast**。

这四个类型转换符中的最后一个是 **reinterpret_cast**。使用这个操作符的类型转换，其的转换结果几乎都是执行期定义（**implementation-defined**）。因此，使用 **reinterpret_casts** 的代码很难移植。

reinterpret_casts 的最普通的用途就是在函数指针类型之间进行转换。例如，假设你有一个函数指针数组：

```
typedef void (*FuncPtr)();       // FuncPtr is 一个指向函数
                                 // 的指针，该函数没有参数
                    // 返回值类型为 void
FuncPtr funcPtrArray[10];        // funcPtrArray 是一个能容纳
                                 // 10 个 FuncPtrs 指针的数组
```

让我们假设你希望（因为某些莫名其妙的原因）把一个指向下面函数的指针存入 **funcPtrArray** 数组：

```
int doSomething();
```

你不能不经过类型转换而直接去做，因为 **doSomething** 函数对于 **funcPtrArray** 数组来说有一个错误的类型。在 **FuncPtrArray** 数组里的函数返回值是 **void** 类型，而 **doSomething** 函数返回值是 **int** 类型。

```
funcPtrArray[0] = &doSomething;      // 错误！类型不匹配
```

reinterpret_cast 可以让你迫使编译器以你的方法去看待它们：

```
funcPtrArray[0] =                    // this compiles
  reinterpret_cast<FuncPtr>(&doSomething);
```

转换函数指针的代码是不可移植的（**C++**不保证所有的函数指针都被用一样的方法表示），在一些情况下这样的转换会产生不正确的结果（参见条款 **M31**），所以你应该避免转换函数指针类型，除非你处于着背水一战和尖刀架喉的危急时刻。一把锋利的刀。一把非常锋利的刀。

如果你使用的编译器缺乏对新的类型转换方式的支持，你可以用传统的类型转换方法代替 **static_cast, const_cast,** 以及 **reinterpret_cast**。也可以用下面的宏替换来模拟新的类型转换语法：

```
#define static_cast(TYPE,EXPR)       ((TYPE)(EXPR))
#define const_cast(TYPE,EXPR)        ((TYPE)(EXPR))
#define reinterpret_cast(TYPE,EXPR)  ((TYPE)(EXPR))
```

你可以象这样使用使用：

```
double result = static_cast(double, firstNumber)/secondNumber;
```

```
update(const_cast(SpecialWidget*, &sw));
funcPtrArray[0] = reinterpret_cast(FuncPtr, &doSomething);
```

这些模拟不会象真实的操作符一样安全，但是当你的编译器可以支持新的的类型转换时，它们可以简化你把代码升级的过程。

没有一个容易的方法来模拟 **dynamic_cast** 的操作，但是很多函数库提供了函数，安全地在派生类与基类之间进行类型转换。如果你没有这些函数而你有必须进行这样的类型转换，你也可以回到 **C** 风格的类型转换方法上，但是这样的话你将不能获知类型转换是否失败。当然，你也可以定义一个宏来模拟 **dynamic_cast** 的功能，就象模拟其它的类型转换一样：

```
#define dynamic_cast(TYPE,EXPR)      (TYPE)(EXPR)
```

请记住，这个模拟并不能完全实现 **dynamic_cast** 的功能，它没有办法知道转换是否失败。

我知道，是的，我知道，新的类型转换操作符不是很美观而且用键盘键入也很麻烦。如果你发现它们看上去实在令人讨厌，**C** 风格的类型转换还可以继续使用并且合法。然而，正是因为新的类型转换符缺乏美感才能使它弥补了在含义精确性和可辨认性上的缺点。并且，使用新类型转换符的程序更容易被解析（不论是对人工还是对于工具程序），它们允许编译器检测出原来不能发现的错误。这些都是放弃 **C** 风格类型转换方法的强有力的理由。还有第三个理由：也许让类型转换符不美观和键入麻烦是一件好事。

3.3 Item M3：不要对数组使用多态

类继承的最重要的特性是你可以通过基类指针或引用来操作派生类。这样的指针或引用具有行为的多态性，就好像它们同时具有多种形态。**C++**允许你通过基类指针和引用来操作派生类数组。不过这根本就不是一个特性，因为这样的代码几乎从不如你所愿地那样运行。

假设你有一个类 **BST**（比如是搜索树对象）和继承自 **BST** 类的派生类 **BalancedBST**：

```
class BST { ... };
class BalancedBST: public BST { ... };
```

在一个真实的程序里，这样的类应该是模板类，但是在这个例子里并不重要，加上模板只会使得代码更难阅读。为了便于讨论，我们假设 **BST** 和 **BalancedBST** 只包含 **int** 类型数据。

有这样一个函数，它能打印出 **BST** 类数组中每一个 **BST** 对象的内容：

```
void printBSTArray(ostream& s,
                   const BST array[],
                   int numElements)
{
  for (int i = 0; i < numElements; ) {
    s << array[i];          //假设 BST 类
```

```
  }                                    //重载了操作符<<
}
```

当你传递给该函数一个含有 **BST** 对象的数组变量时，它能够正常运行：

```
BST BSTArray[10];
...
printBSTArray(cout, BSTArray, 10);          // 运行正常
```

然而，请考虑一下，当你把含有 **BalancedBST** 对象的数组变量传递给 **printBSTArray** 函数时，会产生什么样的后果：

```
BalancedBST bBSTArray[10];
...
printBSTArray(cout, bBSTArray, 10);          // 还会运行正常么？
```

你的编译器将会毫无警告地编译这个函数，但是再看一下这个函数的循环代码：

```
for (int i = 0; i < numElements; ) {
  s << array[i];
}
```

这里的 **array[I]**只是一个指针算法的缩写：它所代表的是***(array)**。我们知道 **array** 是一个指向数组起始地址的指针，但是 **array** 中各元素内存地址与数组的起始地址的间隔究竟有多大呢？它们的间隔是 **i*sizeof(**一个在数组里的对象**)**，因为在 **array** 数组**[0]**到**[I]**间有 **I** 个对象。编译器为了建立正确遍历数组的执行代码，它必须能够确定数组中对象的大小，这对编译器来说是很容易做到的。参数 **array** 被声明为 **BST** 类型，所以 **array** 数组中每一个元素都是 **BST** 类型，因此每个元素与数组起始地址的间隔是 **i*sizeof(BST)**。

至少你的编译器是这么认为的。但是如果你把一个含有 **BalancedBST** 对象的数组变量传递给 **printBSTArray** 函数，你的编译器就会犯错误。在这种情况下，编译器原先已经假设数组中元素与 **BST** 对象的大小一致，但是现在数组中每一个对象大小却与 **BalancedBST** 一致。派生类的长度通常都比基类要长。我们料想 **BalancedBST** 对象长度的比 **BST** 长。如果如此的话，**printBSTArray** 函数生成的指针算法将是错误的，没有人知道如果用 **BalancedBST** 数组来执行 **printBSTArray** 函数将会发生什么样的后果。不论是什么后果都是令人不愉快的。

如果你试图删除一个含有派生类对象的数组，将会发生各种各样的问题。以下是一种你可能采用的但不正确的做法。

```
//删除一个数组，但是首先记录一个删除信息
void deleteArray(ostream& logStream, BST array[])
{
  logStream << "Deleting array at address "
            << static_cast<void*>(array) << '\n';
```

```
delete [] array;
}
BalancedBST *balTreeArray =                    // 建立一个 BalancedBST 对象数组
  new BalancedBST[50];
...
deleteArray(cout, balTreeArray);               // 记录这个删除操作
```

这里面也掩藏着你看不到的指针算法。当一个数组被删除时，每一个数组元素的析构函数也会被调用。当编译器遇到这样的代码：

```
delete [] array;
```

它肯定象这样生成代码：

```
// 以与构造顺序相反的顺序来
// 解构 array 数组里的对象
for ( int i = 数组元素的个数 1; i >= 0;--i)
 {
    array[i].BST::~BST();                      // 调用 array[i]的
  }                                            // 析构函数
```

因为你所编写的循环语句根本不能正确运行，所以当编译成可执行代码后，也不可能正常运行。语言规范中说通过一个基类指针来删除一个含有派生类对象的数组，结果将是不确定的。这实际意味着执行这样的代码肯定不会有什么好结果。多态和指针算法不能混合在一起来用，所以数组与多态也不能用在一起。

值得注意的是如果你不从一个具体类（**concrete classes**）（例如 **BST**）派生出另一个具体类（例如 **BalancedBST**），那么你就不太可能犯这种使用多态性数组的错误。正如条款 **M33** 所解释的，不从具体类派生出具体类有很多好处。我希望你阅读一下条款 **M33** 的内容。

（WQ 加注：VC++中，能够有正确的结果，因为它根本没有数组 new/delete 函数。）

3.4 Item M4：避免无用的缺省构造函数

缺省构造函数（指没有参数的构造函数）在 **C++**语言中是一种让你无中生有的方法。构造函数能初始化对象,而缺省构造函数则可以不利用任何在建立对象时的外部数据就能初始化对象。有时这样的方法是不错的。例如一些行为特性与数字相仿的对象被初始化为空值或不确定的值也是合理的，还有比如链表、哈希表、图等等数据结构也可以被初始化为空容器。

但不是所有的对象都属于上述类型，对于很多对象来说，不利用外部数据进行完全的初始化是不合理的。比如一个没有输入姓名的地址簿对象，就没有任何意义。在一些公司里，所有的设备都必须标有一个公司 **ID** 号码，所以在建立对象以模型化一个设备时，不提供一个合适的 **ID** 号码，所建立的对象就根本没有意义。

在一个完美的世界里，无需任何数据即可建立对象的类可以包含缺省构造函数，而需要数据来建立对象的类则不能包含缺省构造函数。唉！可是我们的现实世界不是完美的，所以我们必须考虑更多的因素。特别是如果一个类没有缺省构造函数，就会存在一些使用上的限制。

请考虑一下有这样一个类，它表示公司的设备，这个类包含一个公司的 ID 代码，这个 ID 代码被强制做为构造函数的参数：

```
class EquipmentPiece {
public:
  EquipmentPiece(int IDNumber);
  ...
};
```

因为 EquipmentPiece 类没有一个缺省构造函数，所以在三种情况下使用它，就会遇到问题。第一中情况是建立数组时。一般来说，没有一种办法能在建立对象数组时给构造函数传递参数。所以在通常情况下，不可能建立 EquipmentPiece 对象数组：

```
EquipmentPiece bestPieces[10];          // 错误！没有正确调用
                                        // EquipmentPiece 构造函数
 EquipmentPiece *bestPieces =
  new EquipmentPiece[10];               // 错误！与上面的问题一样
```

不过还是有三种方法能回避开这个限制。对于使用非堆数组（non-heap arrays）（即不在堆中给数组分配内存。译者注）的一种解决方法是在数组定义时提供必要的参数：

```
int ID1, ID2, ID3, ..., ID10;           // 存储设备 ID 号的
                                        // 变量
...
EquipmentPiece bestPieces[] = {         // 正确，提供了构造
  EquipmentPiece(ID1),                  // 函数的参数
  EquipmentPiece(ID2),
  EquipmentPiece(ID3),
  ...,
  EquipmentPiece(ID10)
};
```

不过很遗憾，这种方法不能用在堆数组(heap arrays)的定义上。

一个更通用的解决方法是利用指针数组来代替一个对象数组：

```
typedef EquipmentPiece* PEP;            //  PEP 指针指向
                                        //一个 EquipmentPiece 对象
```

```
PEP bestPieces[10];                    // 正确，没有调用构造函数
PEP *bestPieces = new PEP[10];         // 也正确
```

在指针数组里的每一个指针被重新赋值，以指向一个不同的 **EquipmentPiece** 对象：

```
for (int i = 0; i < 10; ++i)
  bestPieces[i] = new EquipmentPiece( ID Number );
```

不过这中方法有两个缺点，第一你必须删除数组里每个指针所指向的对象。如果你忘了，就会发生内存泄漏。第二增加了内存分配量，因为正如你需要空间来容纳 **EquipmentPiece** 对象一样，你也需要空间来容纳指针。

如果你为数组分配 **raw memory**，你就可以避免浪费内存。使用 **placement new** 方法（参见条款 M8）在内存中构造 **EquipmentPiece** 对象：

```
// 为大小为 10 的数组 分配足够的内存
// EquipmentPiece 对象; 详细情况请参见条款 M8
// operator new[] 函数
void *rawMemory =
  operator new[](10*sizeof(EquipmentPiece));
// make bestPieces point to it so it can be treated as an
// EquipmentPiece array
EquipmentPiece *bestPieces =
  static_cast<EquipmentPiece*>(rawMemory);
// construct the EquipmentPiece objects in the memory
// 使用"placement new" (参见条款 M8)
for (int i = 0; i < 10; ++i)
  new (&bestPieces[i]) EquipmentPiece( ID Number );
```

注意你仍旧得为每一个 **EquipmentPiece** 对象提供构造函数参数。这个技术（和指针数组的主意一样）允许你在没有缺省构造函数的情况下建立一个对象数组。它没有绕过对构造函数参数的需求，实际上也做不到。如果能做到的话，就不能保证对象被正确初始化。

使用 **placement new** 的缺点除了是大多数程序员对它不熟悉外（能使用它就更难了），还有就是当你不想让它继续存在使用时，必须手动调用数组对象的析构函数，然后调用操作符 **delete[]**来释放 **raw memory**（请再参见条款 M8）：**（WQ 加注，已经有 placement delete/delete []操作符了，它会自动调用析构函数。）**

```
// 以与构造 bestPieces 对象相反的顺序
// 解构它。
for (int i = 9; i >= 0; --i)
```

```
  bestPieces[i].~EquipmentPiece();
// deallocate the raw memory
operator delete[](rawMemory);
```

如果你忘记了这个要求而使用了普通的数组删除方法，那么你程序的运行将是不可预测的。这是因为：直接删除一个不是用 **new** 操作符来分配的内存指针，其结果没有被定义。

```
delete [] bestPieces;                    // 没有定义! bestPieces
                                         //不是用 new 操作符分配的。
```

有关 **new**、**placement new** 和它们如何与构造函数、析构函数一起使用的更多信息，请见条款 M8。

对于类里没有定义缺省构造函数所造成的第二个问题是它们无法在许多基于模板（**template-based**）的容器类里使用。因为实例化一个模板时，模板的类型参数应该提供一个缺省构造函数，这是一个常见的要求。这个要求总是来自于模板内部，被建立的模板参数类型数组里。例如一个数组模板类：

```
template<class T>
class Array {
public:
  Array(int size);
  ...
private:
  T *data;
};
template<class T>
Array<T>::Array(int size)
{
  data = new T[size];                    // 为每个数组元素
  ...                                    //依次调用 T::T()
}
```

在多数情况下，通过仔细设计模板可以杜绝对缺省构造函数的需求。例如标准的 **vector** 模板（生成一个类似于可扩展数组的类）对它的类型参数没有必须有缺省构造函数的要求。不幸的是，很多模板类没有以仔细的态度去设计。这样，没有缺省构造函数的类就不能与许多模板兼容。当 **C++**程序员深入领会了模板设计以后，这样的问题应该不再那么突出了。这会花多长时间，完全在于个人的造化。

最后讲一下在设计虚基类时所面临的要提供缺省构造函数还是不提供缺省构造函数的两难决策。不提供缺省构造函数的虚基类，很难与其进行合作。因为几乎所有的派生类在实

例化时都必须给虚基类构造函数提供参数。这就要求所有由没有缺省构造函数的虚基类继承下来的派生类(无论有多远)都必须知道并理解提供给虚基类构造函数的参数的含义。派生类的作者是不会企盼和喜欢这种规定的。

因为这些强加于没有缺省构造函数的类上的种种限制，一些人认为所有的类都应该有缺省构造函数，即使缺省构造函数没有足够的数据来完整初始化一个对象。比如这个原则的拥护者会这样修改 **EquipmentPiece** 类：

```
class EquipmentPiece {
public:
  EquipmentPiece(  int IDNumber = UNSPECIFIED);
  ...
private:
  static const int   UNSPECIFIED;          // 其值代表 ID 值不确定。
};
```

这允许这样建立 **EquipmentPiece** 对象

```
EquipmentPiece e;                          //这样合法
```

这样的修改使得其他成员函数变得复杂，因为不再能确保 **EquipmentPiece** 对象进行了有意义的初始化。假设它建立一个因没有 **ID** 而没有意义的 **EquipmentPiece** 对象，那么大多数成员函数必须检测 **ID** 是否存在。如果不存在 **ID**，它们将必须指出怎么犯的错误。不过通常不明确应该怎么去做，很多代码的实现什么也没有提供：只是抛出一个异常或调用一个函数终止程序。当这种情形发生时，很难说提供缺省构造函数而放弃了一种保证机制的做法是否能提高软件的总体质量。

提供无意义的缺省构造函数也会影响类的工作效率。如果成员函数必须测试所有的部分是否都被正确地初始化，那么这些函数的调用者就得为此付出更多的时间。而且还得付出更多的代码，因为这使得可执行文件或库变得更大。它们也得在测试失败的地方放置代码来处理错误。如果一个类的构造函数能够确保所有的部分被正确初始化，所有这些弊病都能够避免。缺省构造函数一般不会提供这种保证，所以在它们可能使类变得没有意义时，尽量去避免使用它们。使用这种（没有缺省构造函数的）类的确有一些限制，但是当你使用它时，它也给你提供了一种保证：你能相信这个类被正确地建立和高效地实现。

4. 运算符

运算符重载——你不得不喜欢它们！它们允许给予你的自定义类型有着和 **C++** 内建类型完全相样的语法，更有甚者，它们允许你将强大的能量注入到运算符背后的函数体中，而这是在内建类型上从未听说过的。当然，你能够使得如同“**+**”和“**==**”这样的符号做任何你想做的事，这个事实意味着使用运算符重载你可能写出的程序完全无法理解。**C++** 的老

手知道如何驾驭运算符重载的威力而不至于滑落到“不可理解”的深渊。

遗憾的是，很容易导致滑落。单参数的构造函数和隐式类型转换操作符尤其棘手，因为它们会被调用在没有任何的源代码显示了这样的调用的地方。这会导致程序的行为难于理解。一个不同的问题发生在重载“&&”和“||”这样的运算符时，因为从内建类型到自定义类型的类型转换函数在语法上产生了一个微妙的变化，而这一点非常容易被忽视。最后，许多操作符通过标准的方式彼此的联系在一起，但重载操作符使得改变这种公认的联系成为可能。

在下面的条款中，我集中解释重载的运算符何时并且如何被调用，它们如何运作，它们应该如何彼此联系，以及如何获得这些方面的控制权。有了这一章的信息，你可以和专家一样重载（或不重载）一个运算符。

4.1 Item M5：谨慎定义类型转换函数

C＋＋编译器能够在两种数据类型之间进行隐式转换（implicit conversions），它继承了 C 语言的转换方法，例如允许把 char 隐式转换为 int 和从 short 隐式转换为 double。因此当你把一个 short 值传递给准备接受 double 参数值的函数时，依然可以成功运行。C 中许多这种可怕的转换可能会导致数据的丢失，它们在 C++中依然存在，包括 int 到 short 的转换和 double 到 char 的转换。

你对这些类型转换是无能为力的，因为它们是语言本身的特性。不过当你增加自己的类型时，你就可以有更多的控制力，因为你能选择是否提供函数让编译器进行隐式类型转换。

有两种函数允许编译器进行这些的转换：单参数构造函数（single-argument constructors）和隐式类型转换运算符。单参数构造函数是指只用一个参数即可以调用的构造函数。该函数可以是只定义了一个参数，也可以是虽定义了多个参数但第一个参数以后的所有参数都有缺省值。以下有两个例子：

```
class Name {                              // for names of things
public:
  Name(const string& s);                  // 转换 string 到
                                          // Name
  ...
};
class Rational {                          // 有理数类
public:
  Rational(int numerator = 0,             // 转换 int 到
           int denominator = 1);          // 有理数类
  ...
```

```
};
```

隐式类型转换运算符只是一个样子奇怪的成员函数：**operator** 关键字，其后跟一个类型符号。你不用定义函数的返回类型，因为返回类型就是这个函数的名字。例如为了允许**Rational(**有理数**)**类隐式地转换为 **double** 类型（在用有理数进行混合类型运算时，可能有用），你可以如此声明 **Rational** 类：

```
class Rational {
public:
  ...
  operator double() const;                    // 转换 Rational 类成
};                                            // double 类型
```

在下面这种情况下，这个函数会被自动调用：

```
Rational r(1, 2);                             // r 的值是 1/2

double d = 0.5 * r;                           // 转换 r 到 double,
                                              // 然后做乘法
```

以上这些说明只是一个复习，我真正想说的是为什么你不需要定义各种类型转换函数。

根本问题是当你在不需要使用转换函数时，这些的函数缺却会被调用运行。结果，这些不正确的程序会做出一些令人恼火的事情，而你又很难判断出原因。

让我们首先分析一下隐式类型转换运算符，它们是最容易处理的。假设你有一个如上所述的 **Rational** 类，你想让该类拥有打印有理数对象的功能，就好像它是一个内置类型。因此，你可能会这么写：

```
Rational r(1, 2);
cout << r;                                    // 应该打印出"1/2"
```

再假设你忘了为 **Rational** 对象定义 **operator<<**。你可能想打印操作将失败，因为没有合适的的 **operator<<**被调用。但是你错了。当编译器调用 **operator<<**时，会发现没有这样的函数存在，但是它会试图找到一个合适的隐式类型转换顺序以使得函数调用正常运行。类型转换顺序的规则定义是复杂的，但是在现在这种情况下，编译器会发现它们能调用 **Rational::operator double** 函数来把 **r** 转换为 **double** 类型。所以上述代码打印的结果是一个浮点数，而不是一个有理数。这简直是一个灾难，但是它表明了隐式类型转换的缺点：它们的存在将导致错误的发生。

解决方法是用不使用语法关键字的等同的函数来替代转换运算符。例如为了把 **Rational** 对象转换为 **double**，用 **asDouble** 函数代替 **operator double** 函数：

```
class Rational {
public:
```

```
  ...
  double asDouble() const;                    //转变 Rational
};                                          // 成 double
```

这个成员函数能被显式调用：

```
Rational r(1, 2);

cout << r;                                  // 错误! Rationa 对象没有
                                            // operator<<
cout << r.asDouble();                       // 正确，用 double 类型
                                           //打印 r
```

在多数情况下，这种显式转换函数的使用虽然不方便，但是函数被悄悄调用的情况不再会发生，这点损失是值得的。一般来说，越有经验的 C++程序员就越喜欢避开类型转换运算符。例如在 C++标准库（参见 Effective C＋＋条款 49 和 M35）委员会工作的人员是在此领域最有经验的，他们加在库函数中的 string 类型没有包括隐式地从 string 转换成 C 风格的 char*的功能，而是定义了一个成员函数 c_str 用来完成这个转换，这是巧合么？我看不是。

通过单参数构造函数进行隐式类型转换更难消除。而且在很多情况下这些函数所导致的问题要甚于隐式类型转换运算符。

举一个例子，一个 array 类模板，这些数组需要调用者确定边界的上限与下限：

```
template<class T>
class Array {
public:
  Array(int lowBound, int highBound);
  Array(int size);
  T& operator[](int index);
  ...
};
```

第一个构造函数允许调用者确定数组索引的范围，例如从 10 到 20。它是一个两参数构造函数，所以不能做为类型转换函数。第二个构造函数让调用者仅仅定义数组元素的个数（使用方法与内置数组的使用相似），不过不同的是它能做为类型转换函数使用，能导致无穷的痛苦。

例如比较 Array<int>对象，部分代码如下：

```
bool operator==( const Array<int>& lhs,
                 const Array<int>& rhs);
Array<int> a(10);
```

```
Array<int> b(10);
...
for (int i = 0; i < 10; ++i)
  if (a == b[i]) {                // 哎呦! "a" 应该是 "a[i]"
    do something for when
    a[i] and b[i] are equal;
  }
  else {
    do something for when they're not;
  }
```

我们想用 **a** 的每个元素与 **b** 的每个元素相比较，但是当录入 **a** 时，我们偶然忘记了数组下标。当然我们希望编译器能报出各种各样的警告信息，但是它根本没有。因为它把这个调用看成用 **Array<int>**参数**(**对于 **a)**和 **int(**对于 **b[i])**参数调用 **operator==**函数，然而没有 **operator==**函数是这样的参数类型，我们的编译器注意到它能通过调用 **Array<int>**构造函数能转换 **int** 类型到 **Array<int>**类型，这个构造函数只有一个 **int** 类型的参数。然后编译器如此去编译，生成的代码就象这样：

```
for (int i = 0; i < 10; ++i)
  if (a == static_cast< Array<int> >(b[i]))   ...
```

每一次循环都把 **a** 的内容与一个大小为 **b[i]**的临时数组（内容是未定义的）比较。这不仅不可能以正确的方法运行，而且还是效率低下的。因为每一次循环我们都必须建立和释放 **Array<int>**对象（见条款 **M19**）。

通过不声明运算符（**operator**）的方法**,**可以克服隐式类型转换运算符的缺点，但是单参数构造函数没有那么简单。毕竟，你确实想给调用者提供一个单参数构造函数。同时你也希望防止编译器不加鉴别地调用这个构造函数。幸运的是，有一个方法可以让你鱼肉与熊掌兼得。事实上是两个方法：一是容易的方法，二是当你的编译器不支持容易的方法时所必须使用的方法。

容易的方法是利用一个最新编译器的特性，**explicit** 关键字。为了解决隐式类型转换而特别引入的这个特性，它的使用方法很好理解。构造函数用 **explicit** 声明，如果这样做，编译器会拒绝为了隐式类型转换而调用构造函数。显式类型转换依然合法：

```
template<class T>
class Array {
public:
  ...
  explicit Array(int size);           // 注意使用"explicit"
```

```
  ...
};
Array<int> a(10);                    // 正确, explicit 构造函数
                                     // 在建立对象时能正常使用
Array<int> b(10);                    // 也正确
if (a == b[i]) ...                   // 错误! 没有办法
                                     // 隐式转换
                                     // int 到 Array<int>
if (a == Array<int>(b[i])) ...       // 正确,显式从 int 到
                                     // Array<int>转换
                                     // （但是代码的逻辑
                                     // 不合理）
if (a == static_cast< Array<int> >(b[i])) ...
                                     // 同样正确，同样
                                     // 不合理
if (a == (Array<int>)b[i]) ...       //C 风格的转换也正确，
                                     // 但是逻辑
                                     // 依旧不合理
```

在例子里使用了 **static_cast**（参见条款 **M2**），两个“**>**”字符间的空格不能漏掉，如果这样写语句：

```
if (a == static_cast<Array<int>>(b[i])) ...
```

这是一个不同的含义的语句。因为 **C++**编译器把“**>>**”做为一个符号来解释。在两个“**>**”间没有空格，语句会产生语法错误。

如果你的编译器不支持 **explicit**，你不得不回到不使用成为隐式类型转换函数的单参数构造函数。

我前面说过复杂的规则决定哪一个隐式类型转换是合法的，哪一个是不合法的。这些规则中没有一个转换能够包含用户自定义类型（调用单参数构造函数或隐式类型转换运算符）。你能利用这个规则来正确构造你的类，使得对象能够正常构造，同时去掉你不想要的隐式类型转换。

再来想一下数组模板，你需要用整形变量做为构造函数参数来确定数组大小，但是同时又必须防止从整数类型到临时数组对象的隐式类型转换。你要达到这个目的，先要建立一个新类 **ArraySize**。这个对象只有一个目的就是表示将要建立数组的大小。你必须修改 **Array** 的单参数构造函数，用一个 **ArraySize** 对象来代替 **int**。代码如下：

```
template<class T>
```

```
class Array {
public:
  class ArraySize {                    // 这个类是新的
  public:
    ArraySize(int numElements): theSize(numElements) {}
    int size() const { return theSize; }
  private:
    int theSize;
  };
Array(int lowBound, int highBound);
  Array(ArraySize size);                  // 注意新的声明
...
};
```

这里把 **ArraySize** 嵌套入 **Array** 中，为了强调它总是与 **Array** 一起使用。你也必须声明 **ArraySize** 为公有，为了让任何人都能使用它。

想一下，当通过单参数构造函数定义 **Array** 对象，会发生什么样的事情：

```
Array<int> a(10);
```

你的编译器要求用 **int** 参数调用 **Array<int>**里的构造函数，但是没有这样的构造函数。编译器意识到它能从 **int** 参数转换成一个临时 **ArraySize** 对象，**ArraySize** 对象只是 **Array<int>**构造函数所需要的，这样编译器进行了转换。函数调用（及其后的对象建立）也就成功了。

事实上你仍旧能够安心地构造 **Array** 对象，不过这样做能够使你避免类型转换。考虑一下以下代码：

```
bool operator==( const Array<int>& lhs,
                 const Array<int>& rhs);
Array<int> a(10);
Array<int> b(10);
...
for (int i = 0; i < 10; ++i)
  if (a == b[i]) ...                  // 哎呦! "a" 应该是 "a[i]";
                                      // 现在是一个错误。
```

为了调用 **operator**＝＝函数，编译器要求 **Array<int>**对象在”**==**”右侧,但是不存在一个参数为 **int** 的单参数构造函数。而且编译器无法把 **int** 转换成一个临时 **ArraySize** 对象然后通过这个临时对象建立必须的 **Array<int>**对象，因为这将调用两个用户定义

（user-defined）的类型转换，一个从 int 到 ArraySize，一个从 ArraySize 到 Array<int>。这种转换顺序被禁止的，所以当试图进行比较时编译器肯定会产生错误。

ArraySize 类的使用有些象一个有目的的帮手，这是一个更通用技术的应用实例。类似于 ArraySize 的类经常被称为 proxy classes，因为这样类的每一个对象都为了支持其他对象的工作。ArraySize 对象实际是一个整数类型的替代者，用来在建立 Array 对象时确定数组大小。Proxy 对象能帮你更好地控制软件的在某些方面的行为,否则你就不能控制这些行为，比如在上面的情况里，这种行为是指隐式类型转换，所以它值得你去学习和使用。你可能会问你如何去学习它呢？一种方法是转向条款 M30；它专门讨论 proxy classes。

在你跳到条款 M30 之前，再仔细考虑一下本条款的内容。让编译器进行隐式类型转换所造成的弊端要大于它所带来的好处，所以除非你确实需要，不要定义类型转换函数。

4.2 Item M6：自增(increment)、自减(decrement)操作符前缀形式与后缀形式的区别

很久以前（八十年代），没有办法区分++和--操作符的前缀与后缀调用。这个问题遭到程序员的报怨，于是 C++语言得到了扩展，允许重载 increment 和 decrement 操作符的两种形式。

然而有一个句法上的问题，重载函数间的区别决定于它们的参数类型上的差异，但是不论是 increment 或 decrement 的前缀还是后缀都只有一个参数。为了解决这个语言问题，C++规定后缀形式有一个 int 类型参数，当函数被调用时，编译器传递一个 0 做为 int 参数的值给该函数：

```
class UPInt {                          // "unlimited precision int"
public:
  UPInt& operator++();                 // ++ 前缀
  const UPInt operator++(int);            // ++ 后缀
  UPInt& operator--();                  // -- 前缀
  const UPInt operator--(int);            // -- 后缀
  UPInt& operator+=(int);              // += 操作符，UPInts
                                      // 与 ints 相运算
  ...
};
UPInt i;
++i;                                   // 调用 i.operator++();
i++;                                   // 调用 i.operator++(0);
--i;                                   // 调用 i.operator--();
i--;                                   // 调用 i.operator--(0);
```

这个规范有一些古怪，不过你会习惯的。而尤其要注意的是：这些操作符前缀与后缀形式返回值类型是不同的。前缀形式返回一个引用，后缀形式返回一个 **const** 类型。下面我们将讨论**++**操作符的前缀与后缀形式，这些说明也同样适用于**--**操作符。

从你开始做 **C** 程序员那天开始，你就记住 **increment** 的前缀形式有时叫做“增加然后取回”，后缀形式叫做“取回然后增加”。这两句话非常重要，因为它们是 **increment** 前缀与后缀的形式上的规范。

```
// 前缀形式：增加然后取回值
UPInt& UPInt::operator++()
{
  *this += 1;                                  // 增加
  return *this;                                // 取回值
}
// postfix form: fetch and increment
const UPInt UPInt::operator++(int)
{
  UPInt oldValue = *this;                      // 取回值
  ++(*this);          // 增加
return oldValue;                               // 返回被取回的值
}
```

后缀操作符函数没有使用它的参数。它的参数只是用来区分前缀与后缀函数调用。如果你没有在函数里使用参数，许多编译器会显示警告信息，很令人讨厌。为了避免这些警告信息，一种经常使用的方法时省略掉你不想使用的参数名称；如上所示。

很明显一个后缀 **increment** 必须返回一个对象（它返回的是增加前的值），但是为什么是 **const** 对象呢？假设不是 **const** 对象，下面的代码就是正确的：

```
UPInt i;
i++++;                                          // 两次 increment 后缀
```

这组代码与下面的代码相同：

```
i.operator++(0).operator++(0);
```

很明显，第一个调用的 **operator++**函数返回的对象调用了第二个 **operator++**函数。

有两个理由导致我们应该厌恶上述这种做法，第一是与内置类型行为不一致。当设计一个类遇到问题时，一个好的准则是使该类的行为与 **int** 类型一致。而 **int** 类型不允许连续进行两次后缀 **increment**：

```
int i;
i++++;                                            // 错误!
```

第二个原因是使用两次后缀 increment 所产生的结果与调用者期望的不一致。如上所示，第二次调用 operator++改变的值是第一次调用返回对象的值，而不是原始对象的值。因此如果：

```
i++++;
```

是合法的，i 将仅仅增加了一次。这与人的直觉相违背，使人迷惑（对于 int 类型和 UPInt 都是一样），所以最好禁止这么做。

C++禁止 int 类型这么做，同时你也必须禁止你自己写的类有这样的行为。最容易的方法是让后缀 increment 返回 const 对象。当编译器遇到这样的代码：

```
i++++;                                    // same as
i.operator++(0).operator++(0);
```

它发现从第一个 operator++函数返回的 const 对象又调用 operator++函数，然而这个函数是一个 non－const 成员函数，所以 const 对象不能调用这个函数。如果你原来想过让一个函数返回 const 对象没有任何意义，现在你就知道有时还是有用的，后缀 increment 和 decrement 就是例子。（更多的例子参见 Effective C++ 条款 21）

如果你很关心效率问题，当你第一次看到后缀 increment 函数时，你可能觉得有些问题。这个函数必须建立一个临时对象以做为它的返回值，（参见条款 M19），上述实现代码建立了一个显示的临时对象（oldValue），这个临时对象必须被构造并在最后被析构。前缀 increment 函数没有这样的临时对象。由此得出一个令人惊讶的结论，如果仅为了提高代码效率，UPInt 的调用者应该尽量使用前缀 increment，少用后缀 increment，除非确实需要使用后缀 increment。让我们明确一下，当处理用户定义的类型时，尽可能地使用前缀 increment，因为它的效率较高。

我们再观察一下后缀与前缀 increment 操作符。它们除了返回值不同外，所完成的功能是一样的，即值加一。简而言之，它们被认为功能一样。那么你如何确保后缀 increment 和前缀 increment 的行为一致呢？当不同的程序员去维护和升级代码时，有什么能保证它们不会产生差异？除非你遵守上述代码里的原则，这才能得到确保。这个原则是后缀 increment 和 decrement 应该根据它们的前缀形式来实现。你仅仅需要维护前缀版本，因为后缀形式自动与前缀形式的行为一致。

正如你所看到的，掌握前缀和后缀 increment 和 decrement 是容易的。一旦了解了他们正确的返回值类型以及后缀操作符应该以前缀操作符为基础来实现的规则，就足够了。

4.3 Item M7：不要重载“&&”，“||”，或“,”

与 C 一样，C++使用布尔表达式短路求值法(short-circuit evaluation)。这表示一旦确定了布尔表达式的真假值，即使还有部分表达式没有被测试，布尔表达式也停止运算。例如：

```
char *p;
...
if ((p != 0) && (strlen(p) > 10)) ...
```

这里不用担心当 **p** 为空时 **strlen** 无法正确运行，因为如果 **p** 不等于 **0** 的测试失败，**strlen** 不会被调用。同样：

```
int rangeCheck(int index)
{
if ((index < lowerBound) || (index > upperBound)) ...
...
}
```

如果 **index** 小于 **lowerBound**，它不会与 **upperBound** 进行比较。

很早以前，上述行为特性就被反复灌输给 **C** 和 **C++**的程序员，所以他们都知道该特性。而且他们也依赖于短路求值法来写程序。例如在上述第一个代码中，当 **p** 为空指针时确保 **strlen** 不会被调用是很重要的，因为 **C++**标准说**(**正如 **C** 标准所说**)**用空指针调用 **strlen**，结果不确定。

C++允许根据用户定义的类型，来定制**&&**和**||**操作符。方法是重载函数 **operator&&** 和 **operator||**，你能在全局重载或每个类里重载。然而如果你想使用这种方法，你必须知道你正在极大地改变游戏规则。因为你以函数调用法替代了短路求值法。也就是说如果你重载了操作符**&&**，对于你来说代码是这样的：

```
if (expression1 && expression2) ...
```

对于编译器来说，等同于下面代码之一：

```
if (expression1.operator&&(expression2)) ...
                                // when operator&& is a
                                // member function
if (operator&&(expression1, expression2)) ...
                                // when operator&& is a
                                // global function
```

这好像没有什么不同，但是函数调用法与短路求值法是绝对不同的。首先当函数被调用时，需要运算其所有参数，所以调用函数 **functions operator&&** 和 **operator||**时，两个参数都需要计算，换言之，没有采用短路计算法。第二是 **C++**语言规范没有定义函数参数的计算顺序，所以没有办法知道表达式 **1** 与表达式 **2** 哪一个先计算。完全可能与具有从左参数到右参数计算顺序的短路计算法相反。

因此如果你重载**&&**或**||**，就没有办法提供给程序员他们所期望和使用的行为特性，所以不要重载**&&**和**||**。

同样的理由也适用于逗号操作符，但是在我们深入研究它之前，我还是暂停一下，让你不要太惊讶，“逗号操作符？哪有逗号操作符？”确实存在。逗号操作符用于组成表达式，你经常在 **for** 循环的更新部分（**update part**）里遇见它。例如下面来源于 **Kernighan's and Ritchie's** 经典书籍 **The C Programming Language** 第二版**(Prentice-Hall, 1988)**的函数：

```
// reverse string s in place
void reverse(char s[])
{
    for (int i = 0, j = strlen(s)-1;
         i < j;
         ++i, --j)          // 啊! 逗号操作符!
    {
      int c = s[i];
      s[i] = s[j];
      s[j] = c;
    }
}
```

在 **for** 循环的最后一个部分里，**i** 被增加同时 **j** 被减少。在这里使用逗号很方便，因为在最后一个部分里只能使用一个表达式，分开表达式来改变 **i** 和 **j** 的值是不合法的。

对于内建类型**&&**和**||**，**C++**有一些规则来定义它们如何运算。与此相同，也有规则来定义逗号操作符的计算方法。一个包含逗号的表达式首先计算逗号左边的表达式，然后计算逗号右边的表达式；整个表达式的结果是逗号右边表达式的值。所以在上述循环的最后部分里，编译器首先计算**++i**，然后是—**j**，逗号表达式的结果是**--j**。

也许你想为什么你需要知道这些内容呢？因为你需要模仿这个行为特性，如果你想大胆地写自己的逗号操作符函数。不幸的是你无法模仿。

如果你写一个非成员函数 **operator**，你不能保证左边的表达式先于右边的表达式计算，因为函数（**operator**）调用时两个表达式做为参数被传递出去。但是你不能控制函数参数的计算顺序。所以非成员函数的方法绝对不行。

剩下的只有写成员函数 **operator** 的可能性了。即使这里你也不能依靠于逗号左边表达式先被计算的行为特性，因为编译器不一定必须按此方法去计算。因此你不能重载逗号操作符，保证它的行为特性与其被料想的一样。重载它是完全轻率的行为。

你可能正在想这个重载恶梦究竟有没有完。毕竟如果你能重载逗号操作符，你还有什么不能重载的呢？正如显示的，存在一些限制，你不能重载下面的操作符：

```
.              .*              ::              ?:
new          delete        sizeof      typeid
```

static_cast dynamic_cast const_cast reinterpret_cast

你能重载：

operator new operator delete

operator new[] operator delete[]

+ - * / % ^ & | ~

! = < > += -= *= /= %=

^= &= |= << >> >>= <<= == !=

<= >= && || ++ -- , ->* ->

() []

（有关 new 和 delete 还有 operator new, operator delete, operator new[], and operator delete[]的信息参见条款 M8）

当然能重载这些操作符不是去重载的理由。操作符重载的目的是使程序更容易阅读，书写和理解，而不是用你的知识去迷惑其他人。如果你没有一个好理由重载操作符，就不要重载。在遇到&&, ||, 和 ,时，找到一个好理由是困难的，因为无论你怎么努力，也不能让它们的行为特性与所期望的一样。

4.4 Item M8：理解各种不同含义的 new 和 delete

人们有时好像喜欢故意使 C++语言的术语难以理解。比如说 new 操作符（new operator）和 new 操作（operator new）的区别。

当你写这样的代码：

```
string *ps = new string("Memory Management");
```

你使用的 new 是 new 操作符。这个操作符就象 sizeof 一样是语言内置的，你不能改变它的含义，它的功能总是一样的。它要完成的功能分成两部分。第一部分是分配足够的内存以便容纳所需类型的对象。第二部分是它调用构造函数初始化内存中的对象。new 操作符总是做这两件事情，你不能以任何方式改变它的行为。

你所能改变的是如何为对象分配内存。new 操作符调用一个函数来完成必需的内存分配，你能够重写或重载这个函数来改变它的行为。new 操作符为分配内存所调用函数的名字是 operator new。

函数 operator new 通常这样声明：

```
void * operator new(size_t size);
```

返回值类型是 void*，因为这个函数返回一个未经处理（raw）的指针，未初始化的内存。（如果你喜欢，你能写一种 operator new 函数，在返回一个指针之前能够初始化内存以存储一些数值，但是一般不这么做。）参数 size_t 确定分配多少内存。你能增加额外的参数重载函数 operator new，但是第一个参数类型必须是 size_t。（有关 operator new 更多的

信息参见 Effective C++ 条款 8 至条款 10。）

你一般不会直接调用 operator new，但是一旦这么做，你可以象调用其它函数一样调用它：

```
void *rawMemory = operator new(sizeof(string));
```

操作符 operator new 将返回一个指针，指向一块足够容纳一个 string 类型对象的内存。

就象 malloc 一样，operator new 的职责只是分配内存。它对构造函数一无所知。operator new 所了解的是内存分配。把 operator new 返回的未经处理的指针传递给一个对象是 new 操作符的工作。当你的编译器遇见这样的语句：

```
string *ps = new string("Memory Management");
```

它生成的代码或多或少与下面的代码相似（更多的细节见 Effective C++条款 8 和条款 10，还有我的文章 Counting object 里的注释。）：

```
void *memory =                          // 得到未经处理的内存
  operator new(sizeof(string));         // 为 String 对象
call string::string("Memory Management") //初始化
on *memory;                             // 内存中
                                        // 的对象
string *ps =                            // 是 ps 指针指向
  static_cast<string*>(memory);         // 新的对象
```

注意第二步包含了构造函数的调用，你做为一个程序员被禁止这样去做。你的编译器则没有这个约束，它可以做它想做的一切。因此如果你想建立一个堆对象就必须用 new 操作符，不能直接调用构造函数来初始化对象。

I placement new

有时你确实想直接调用构造函数。在一个已存在的对象上调用构造函数是没有意义的，因为构造函数用来初始化对象，而一个对象仅仅能在给它初值时被初始化一次。但是有时你有一些已经被分配但是尚未处理的(raw)内存，你需要在这些内存中构造一个对象。你可以使用一个特殊的 operator new ，它被称为 placement new。

下面的例子是 placement new 如何使用，考虑一下：

```
class Widget {
public:
  Widget(int widgetSize);
  ...
};
Widget * constructWidgetInBuffer(void *buffer,
                                 int widgetSize)
```

```
{
  return new (buffer) Widget(widgetSize);
}
```

这个函数返回一个指针，指向一个 **Widget** 对象，对象在转递给函数的 **buffer** 里分配。当程序使用共享内存或 **memory-mapped I/O** 时这个函数可能有用，因为在这样程序里对象必须被放置在一个确定地址上或一块被例程分配的内存里。（参见条款 **M4,**一个如何使用 **placement new** 的一个不同例子。）

在 **constructWidgetInBuffer** 里面，返回的表达式是：

```
new (buffer) Widget(widgetSize)
```

这初看上去有些陌生,但是它是 **new** 操作符的一个用法,需要使用一个额外的变量(**buffer**)，当 **new** 操作符隐含调用 **operator new** 函数时，把这个变量传递给它。被调用的 **operator new** 函数除了待有强制的参数 **size_t** 外，还必须接受 **void***指针参数，指向构造对象占用的内存空间。这个 **operator new** 就是 **placement new**，它看上去象这样：

```
void * operator new(size_t, void *location)
{
  return location;
}
```

这可能比你期望的要简单,但是这就是 **placement new** 需要做的事情。毕竟 **operator new** 的目的是为对象分配内存然后返回指向该内存的指针。在使用 **placement new** 的情况下，调用者已经获得了指向内存的指针，因为调用者知道对象应该放在哪里。**placement new** 必须做的就是返回转递给它的指针。（没有用的（但是强制的）参数 **size_t** 没有名字，以防止编译器发出警告说它没有被使用；见条款 **M6**。） **placement new** 是标准 **C++**库的一部分（见 **Effective C++** 条款 **49**）。为了使用 **placement new**，你必须使用语句**#include <new>**（或者如果你的编译器还不支持这新风格的头文件名**(**再参见 **Effective C++** 条款 **49)**，**<new.h>**）。

让我们从 **placement new** 回来片刻，看看 **new** 操作符（**new operator**）与 **operator new** 的关系，你想在堆上建立一个对象，应该用 **new** 操作符。它既分配内存又为对象调用构造函数。如果你仅仅想分配内存，就应该调用 **operator new** 函数；它不会调用构造函数。如果你想定制自己的在堆对象被建立时的内存分配过程，你应该写你自己的 **operator new** 函数，然后使用 **new** 操作符，**new** 操作符会调用你定制的 **operator new**。如果你想在一块已经获得指针的内存里建立一个对象，应该用 **placement new**。

（有关更多的不同的 **new** 与 **delete** 的观点参见 **Effective C++** 条款 **7** 和我的文章 **Counting objects**。）

I Deletion and Memory Deallocation

为了避免内存泄漏，每个动态内存分配必须与一个等同相反的 **deallocation** 对应。函数 **operator delete** 与 **delete** 操作符的关系与 **operator new** 与 **new** 操作符的关系一样。当你看到这些代码：

```
string *ps;
...
delete ps;                          // 使用 delete 操作符
```

你的编译器会生成代码来析构对象并释放对象占有的内存。

Operator delete 用来释放内存，它被这样声明：

```
void operator delete(void *memoryToBeDeallocated);
```

因此，

```
delete ps;
```

导致编译器生成类似于这样的代码：

```
ps->~string();                      // call the object's dtor
operator delete(ps);                // deallocate the memory
                                    // the object occupied
```

这有一个隐含的意思是如果你只想处理未被初始化的内存，你应该绕过 **new** 和 **delete** 操作符，而调用 **operator new** 获得内存和 **operator delete** 释放内存给系统：

```
void *buffer =                      // 分配足够的
  operator new(50*sizeof(char));      // 内存以容纳 50 个 char
                                  //没有调用构造函数
...
operator delete(buffer);              // 释放内存
                                  // 没有调用析构函数
```

这与在 **C** 中调用 **malloc** 和 **free** 等同。

如果你用 **placement new** 在内存中建立对象，你应该避免在该内存中用 **delete** 操作符。因为 **delete** 操作符调用 **operator delete** 来释放内存，但是包含对象的内存最初不是被 **operator new** 分配的，**placement new** 只是返回转递给它的指针。谁知道这个指针来自何方？而你应该显式调用对象的析构函数来解除构造函数的影响：

```
// 在共享内存中分配和释放内存的函数
void * mallocShared(size_t size);
void freeShared(void *memory);
void *sharedMemory = mallocShared(sizeof(Widget));
Widget *pw =                                    // 如上所示，
  constructWidgetInBuffer(sharedMemory, 10);    // 使用
```

```
                                        // placement new
...
delete pw;              // 结果不确定! 共享内存来自
                        // mallocShared, 而不是 operator new
pw->~Widget();          // 正确。 析构 pw 指向的 Widget,
                        // 但是没有释放
                        //包含 Widget 的内存
freeShared(pw);         // 正确。 释放 pw 指向的共享内存
                        // 但是没有调用析构函数
```

如上例所示，如果传递给 **placement new** 的 **raw** 内存是自己动态分配的（通过一些不常用的方法），如果你希望避免内存泄漏，你必须释放它。（参见我的文章 **Counting objects** 里面关于 **placement delete** 的注释。）

I Arrays

到目前为止一切顺利，但是还得接着走。到目前为止我们所测试的都是一次建立一个对象。怎样分配数组？会发生什么？

```
string *ps = new string[10];          // allocate an array of
                                      // objects
```

被使用的 **new** 仍然是 **new** 操作符，但是建立数组时 **new** 操作符的行为与单个对象建立有少许不同。第一是内存不再用 **operator new** 分配，代替以等同的数组分配函数，叫做 **operator new[]**（经常被称为 **array new**）。它与 **operator new** 一样能被重载。这就允许你控制数组的内存分配，就象你能控制单个对象内存分配一样（但是有一些限制性说明，参见 **Effective C++** 条款 **8**）。

（**operator new[]**对于 **C++**来说是一个比较新的东西，所以你的编译器可能不支持它。如果它不支持，无论在数组中的对象类型是什么，全局 **operator new** 将被用来给每个数组分配内存。在这样的编译器下定制数组内存分配是困难的，因为它需要重写全局 **operator new**。这可不是一个能轻易接受的任务。缺省情况下，全局 **operator new** 处理程序中所有的动态内存分配，所以它行为的任何改变都将有深入和普遍的影响。而且全局 **operator new** 有一个正常的签名（**normal signature**）**(**也就单一的参数 **size_t**，参见 **Effective C++**条款 **9)**，所以如果你决定用自己的方法声明它，你立刻使你的程序与其它库不兼容（参见条款 **M27**）基于这些考虑，在缺乏 **operator new[]**支持的编译器里为数组定制内存管理不是一个合理的设计。）

第二个不同是 **new** 操作符调用构造函数的数量。对于数组，在数组里的每一个对象的构造函数都必须被调用：

```
string *ps =                  // 调用 operator new[]为 10 个
```

```
new string[10];          // string对象分配内存，
                         // 然后对每个数组元素调用
                         // string对象的缺省构造函数。
```

同样当**delete**操作符用于数组时，它为每个数组元素调用析构函数，然后调用**operator delete**来释放内存。

就象你能替换或重载**operator delete**一样，你也替换或重载**operator delete[]**。在它们重载的方法上有一些限制。请参考优秀的**C++**教材。（有关优秀的**C++**教材的信息，参见本书**285**页的推荐）

new和**delete**操作符是内置的，其行为不受你的控制，凡是它们调用的内存分配和释放函数则可以控制。当你想定制**new**和**delete**操作符的行为时，请记住你不能真的做到这一点。你只能改变它们为完成它们的功能所采取的方法，而它们所完成的功能则被语言固定下来，不能改变。（**You can modify how they do what they do, but what they do is fixed by the language**）

5. 异常

C++新增的异常（**exception**）机制改变了某些事情，这种改变是深刻的，彻底的，可能是令人不舒服的。例如：使用未经处理的或原始的指针变得很危险；资源泄漏的可能性增加了；写出具有你希望的行为的构造函数与析构函数变得更加困难。特别小心防止程序执行时突然崩溃。执行程序和库程序尺寸增加了，同时运行速度降低了。

这就使我们所知道的事情。很多使用**C++**的人都不知道在程序中使用异常，大多数人不知道如何正确使用它。在异常被抛出后，使软件的行为具有可预测性和可靠性，在众多方法中至今也没有一个一致的方法能做到这点。（为了深刻了解这个问题，参见**Tom Cargill**写的**Exception Handling: A False Sense of Security**。有关这些问题的进展情况的信息，参见**Jack Reeves**写的**Coping with Exceptions**和**Herb Sutter**写的**Exception-Safe Generic Containers**。）

我们知道：程序能够在存在异常的情况下正常运行是因为它们按照要求进行了设计，而不是因为巧合。异常安全（**Exception-safe**）的程序不是偶然建立的。一个没有按照要求进行设计的程序在存在异常的情况下运行正常的概率与一个没有按照多线程要求进行设计的程序在多线程的环境下运行正常的概率相同，概率为**0**。

为什么使用异常呢？自从**C**语言被发明初来，**C**程序员就满足于使用错误代码（**Error code**），所以为什么还要弄来异常呢，特别是如果异常如我上面所说的那样存在着问题。答案是简单的：异常不能被忽略。如果一个函数通过设置一个状态变量或返回错误代码来表示一个异常状态，没有办法保证函数调用者将一定检测变量或测试错误代码。结果程序会从它遇到的异常状态继续运行，异常没有被捕获，程序立即会终止执行。

C 程序员能够仅通过 setjmp 和 longjmp 来完成与异常处理相似的功能。但是当 longjmp 在 C++中使用时，它存在一些缺陷，当它调整堆栈时不能对局部对象调用析构函数。（WQ 加注，VC＋＋能保证这一点，但不要依赖这一点。）而大多数 C＋＋程序员依赖于这些析构函数的调用，所以 setjmp 和 longjmp 不能够替换异常处理。如果你需要一个方法，能够通知不可被忽略的异常状态，并且搜索栈空间（searching the stack）以便找到异常处理代码时，你还得确保局部对象的析构函数必须被调用，这时你就需要使用 C++的异常处理。

因为我们已经对使用异常处理的程序设计有了很多了解，下面这些条款仅是一个对于写出异常安全（Exception-safe）软件的不完整的指导。然而它们给任何在 C++中使用异常处理的人介绍了一些重要思想。通过留意下面这些指导，你能够提高自己软件的正确性，强壮性和高效性，并且你将回避开许多在使用异常处理时经常遇到的问题。

5.1 Item M9：使用析构函数防止资源泄漏

对指针说再见。必须得承认：你永远都不会喜欢使用指针。

Ok，你不用对所有的指针说再见，但是你需要对用来操纵局部资源（local resources）的指针说再见。假设，你正在为一个小动物收容所编写软件，小动物收容所是一个帮助小狗小猫寻找主人的组织。每天收容所建立一个文件，包含当天它所管理的收容动物的资料信息，你的工作是写一个程序读出这些文件然后对每个收容动物进行适当的处理（appropriate processing）。

完成这个程序一个合理的方法是定义一个抽象类，ALA（"Adorable Little Animal"），然后为小狗和小猫建立派生类。一个虚拟函数 processAdoption 分别对各个种类的动物进行处理：

```
 class ALA {
public:
  virtual void processAdoption() = 0;
  ...
};
class Puppy: public ALA {
public:
  virtual void processAdoption();
  ...
};
class Kitten: public ALA {
public:
  virtual void processAdoption();
```

```
  ...
};
```

你需要一个函数从文件中读去信息，然后根据文件中的信息产生一个 **puppy**（小狗）对象或者 **kitten(**小猫**)**对象。这个工作非常适合于虚拟构造器（**virtual constructor**），在条款 **M25** 详细描述了这种函数。为了完成我们的目标，我们这样声明函数：

```
// 从 s 中读去动物信息，然后返回一个指针
// 指向新建立的某种类型对象
ALA * readALA(istream& s);
```

你的程序的关键部分就是这个函数，如下所示：

```
void processAdoptions(istream& dataSource)
{
  while (dataSource) {                    // 还有数据时,继续循环
    ALA *pa = readALA(dataSource);        //得到下一个动物
    pa->processAdoption();                //处理收容动物
    delete pa;                            //删除 readALA 返回的对象
  }
}
```

这个函数循环遍历 **dataSource** 内的信息，处理它所遇到的每个项目。唯一要记住的一点是在每次循环结尾处删除 **pa**。这是必须的，因为每次调用 **readALA** 都建立一个堆对象。如果不删除对象，循环将产生资源泄漏。

现在考虑一下，如果 **pa->processAdoption** 抛出了一个异常，将会发生什么？**processAdoptions** 没有捕获异常，所以异常将传递给 **processAdoptions** 的调用者。传递中，**processAdoptions** 函数中的调用 **pa->processAdoption** 语句后的所有语句都被跳过，这就是说 **pa** 没有被删除。结果，任何时候 **pa->processAdoption** 抛出一个异常都会导致 **processAdoptions** 内存泄漏。

堵塞泄漏很容易 ：

```
void processAdoptions(istream& dataSource)
{
  while (dataSource) {
    ALA *pa = readALA(dataSource);
  try {
      pa->processAdoption();
  }
  catch (...) {               // 捕获所有异常
```

```
    delete pa;                  // 避免内存泄漏
                                // 当异常抛出时
    throw;                      // 传送异常给调用者
  }
  delete pa;                    // 避免资源泄漏
}                               // 当没有异常抛出时
}
```

但是你必须用 **try** 和 **catch** 对你的代码进行小改动。更重要的是你必须写双份清除代码，一个为正常的运行准备，一个为异常发生时准备。在这种情况下，必须写两个 **delete** 代码。象其它重复代码一样，这种代码写起来令人心烦又难于维护，而且它看上去好像存在着问题。不论我们是让 **processAdoptions** 正常返回还是抛出异常，我们都需要删除 **pa**，所以为什么我们必须要在多个地方编写删除代码呢？

（WQ 加注，VC＋＋支持 try…catch…final 结构的 SEH。）

我们可以把总被执行的清除代码放入 **processAdoptions** 函数内的局部对象的析构函数里，这样可以避免重复书写清除代码。因为当函数返回时局部对象总是被释放，无论函数是如何退出的。（仅有一种例外就是当你调用 **longjmp** 时。**Longjmp** 的这个缺点是 **C++**率先支持异常处理的主要原因）

具体方法是用一个对象代替指针 **pa**，这个对象的行为与指针相似。当 **pointer-like** 对象（类指针对象）被释放时，我们能让它的析构函数调用 **delete**。替代指针的对象被称为 **smart pointers**（灵巧指针），参见条款 **M28** 的解释，你能使得 **pointer-like** 对象非常灵巧。在这里，我们用不着这么聪明的指针，我们只需要一个 **pointer-lik** 对象，当它离开生存空间时知道删除它指向的对象。

写出这样一个类并不困难，但是我们不需要自己去写。标准 **C++**库函数包含一个类模板，叫做 **auto_ptr**，这正是我们想要的。每一个 **auto_ptr** 类的构造函数里，让一个指针指向一个堆对象（**heap object**），并且在它的析构函数里删除这个对象。下面所示的是 **auto_ptr** 类的一些重要的部分：

```
template<class T>
class auto_ptr {
public:
  auto_ptr(T *p = 0): ptr(p) {}         // 保存 ptr，指向对象
  ~auto_ptr() { delete ptr; }           // 删除 ptr 指向的对象
private:
  T *ptr;                               // raw ptr to object
};
```

auto_ptr 类的完整代码是非常有趣的，上述简化的代码实现不能在实际中应用。（我们至少必须加上拷贝构造函数，赋值 operator 和将在条款 M28 讲述的 pointer-emulating 函数），但是它背后所蕴含的原理应该是清楚的：用 auto_ptr 对象代替 raw 指针，你将不再为堆对象不能被删除而担心，即使在抛出异常时，对象也能被及时删除。(因为 auto_ptr 的析构函数使用的是单对象形式的 delete，所以 auto_ptr 不能用于指向对象数组的指针。如果想让 auto_ptr 类似于一个数组模板，你必须自己写一个。在这种情况下，用 vector 代替 array 可能更好。)

使用 auto_ptr 对象代替 raw 指针，processAdoptions 如下所示：

```
void processAdoptions(istream& dataSource)
{
  while (dataSource) {
    auto_ptr<ALA> pa(readALA(dataSource));
    pa->processAdoption();
  }
}
```

这个版本的processAdoptions在两个方面区别于原来的processAdoptions函数。第一，pa 被声明为一个 auto_ptr<ALA>对象，而不是一个 raw ALA*指针。第二，在循环的结尾没有 delete 语句。其余部分都一样，因为除了析构的方式，auto_ptr 对象的行为就象一个普通的指针。是不是很容易。

隐藏在 auto_ptr 后的思想是：用一个对象存储需要被自动释放的资源，然后依靠对象的析构函数来释放资源，这种思想不只是可以运用在指针上，还能用在其它资源的分配和释放上。想一下这样一个在 GUI 程序中的函数，它需要建立一个 window 来显式一些信息：

```
// 这个函数会发生资源泄漏，如果一个异常抛出
void displayInfo(const Information& info)
{
  WINDOW_HANDLE w(createWindow());
  在 w 对应的 window 中显式信息
  destroyWindow(w);
}
```

很多 window 系统有 C—like 接口，使用象 like createWindow 和 destroyWindow 函数来获取和释放 window 资源。如果在 w 对应的 window 中显示信息时，一个异常被抛出，w 所对应的 window 将被丢失，就象其它动态分配的资源一样。

解决方法与前面所述的一样，建立一个类，让它的构造函数与析构函数来获取和释放资源：

```
//一个类，获取和释放一个 window 句柄
class WindowHandle {
public:
   WindowHandle(WINDOW_HANDLE handle): w(handle) {}
  ~WindowHandle() { destroyWindow(w); }
   operator WINDOW_HANDLE() { return w; }        // see below
private:
  WINDOW_HANDLE w;
  // 下面的函数被声明为私有，防止建立多个 WINDOW_HANDLE 拷贝
  //有关一个更灵活的方法的讨论请参见条款 M28。
  WindowHandle(const WindowHandle&);
  WindowHandle& operator=(const WindowHandle&);
};
```

这看上去有些象 **auto_ptr**，只是赋值操作与拷贝构造被显式地禁止（参见条款 **M27**），有一个隐含的转换操作能把 **WindowHandle** 转换为 **WINDOW_HANDLE**。这个能力对于使用 **WindowHandle** 对象非常重要，因为这意味着你能在任何地方象使用 **raw WINDOW_HANDLE** 一样来使用 **WindowHandle**。（参见条款 **M5** ，了解为什么你应该谨慎使用隐式类型转换操作）

通过给出的 **WindowHandle** 类，我们能够重写 **displayInfo** 函数，如下所示：

```
// 如果一个异常被抛出，这个函数能避免资源泄漏
void displayInfo(const Information& info)
{
  WindowHandle w(createWindow());
  在 w 对应的 window 中显式信息;
}
```

即使一个异常在 **displayInfo** 内被抛出，被 **createWindow** 建立的 **window** 也能被释放。

资源应该被封装在一个对象里，遵循这个规则，你通常就能避免在存在异常环境里发生资源泄漏。但是如果你正在分配资源时一个异常被抛出，会发生什么情况呢？例如当你正处于 **resource-acquiring** 类的构造函数中。还有如果这样的资源正在被释放时，一个异常被抛出，又会发生什么情况呢？构造函数和析构函数需要特殊的技术。你能在条款 **M10** 和条款 **M11** 中获取有关的知识。

5.2 Item M10：在构造函数中防止资源泄漏

如果你正在开发一个具有多媒体功能的通讯录程序。这个通讯录除了能存储通常的文字信息如姓名、地址、电话号码外，还能存储照片和声音（可以给出他们名字的正确发音）。

为了实现这个通信录，你可以这样设计：

```
class Image {                          // 用于图像数据
public:
  Image(const string& imageDataFileName);
  ...
};

class AudioClip {                      // 用于声音数据
public:
  AudioClip(const string& audioDataFileName);
  ...
};

class PhoneNumber {          ... };   // 用于存储电话号码
class BookEntry {                      // 通讯录中的条目
public:

  BookEntry(const string& name,
            const string& address = "",
            const string& imageFileName = "",
            const string& audioClipFileName = "");
  ~BookEntry();
// 通过这个函数加入电话号码
  void addPhoneNumber(const PhoneNumber& number);
...
private:
  string theName;                  // 人的姓名
  string theAddress;               // 他们的地址
  list<PhoneNumber> thePhones;     // 他的电话号码
  Image *theImage;                 // 他们的图像
  AudioClip *theAudioClip;         // 他们的一段声音片段
};
```

通讯录的每个条目都有姓名数据，所以你需要带有参数的构造函数（参见条款 **M3**），不过其它内容（地址、图像和声音的文件名）都是可选的。注意应该使用链表类（**list**）存储

电话号码，这个类是标准 C++类库（STL）中的一个容器类（container classes）。（参见 Effective C++条款 49 和本书条款 M35）

编写 BookEntry 构造函数和析构函数，有一个简单的方法是：

```
BookEntry::BookEntry(const string& name,
                     const string& address,
                     const string& imageFileName,
                     Const string& audioClipFileName)
: theName(name), theAddress(address),
  theImage(0), theAudioClip(0)
{
  if (imageFileName != "") {
    theImage = new Image(imageFileName);
  }
  if (audioClipFileName != "") {
    theAudioClip = new AudioClip(audioClipFileName);
  }
}
BookEntry::~BookEntry()
{
  delete theImage;
  delete theAudioClip;
}
```

构造函数把指针 theImage 和 theAudioClip 初始化为空，然后如果其对应的构造函数参数不是空，就让这些指针指向真实的对象。析构函数负责删除这些指针，确保 BookEntry 对象不会发生资源泄漏。因为 C++确保删除空指针是安全的，所以 BookEntry 的析构函数在删除指针前不需要检测这些指针是否指向了某些对象。

看上去好像一切良好，在正常情况下确实不错，但是在非正常情况下（例如在有异常发生的情况下）它们恐怕就不会良好了。

请想一下如果 BookEntry 的构造函数正在执行中，一个异常被抛出，会发生什么情况呢？：

```
if (audioClipFileName != "") {
  theAudioClip = new AudioClip(audioClipFileName);
}
```

一个异常被抛出，可以是因为 operator new（参见条款 M8）不能给 AudioClip 分配足

够的内存，也可以因为 **AudioClip** 的构造函数自己抛出一个异常。不论什么原因，如果在 **BookEntry** 构造函数内抛出异常，这个异常将传递到建立 **BookEntry** 对象的地方（在构造函数体的外面。 译者注）。

现在假设建立 **theAudioClip** 对象建立时，一个异常被抛出（而且传递程序控制权到 **BookEntry** 构造函数的外面），那么谁来负责删除 **theImage** 已经指向的对象呢？答案显然应该是由 **BookEntry** 来做，但是这个想当然的答案是错的。**~BookEntry()**根本不会被调用，永远不会。

C++仅仅能删除被完全构造的对象（**fully contructed objects**），只有一个对象的构造函数完全运行完毕，这个对象才被完全地构造。所以如果一个 **BookEntry** 对象 **b** 做为局部对象建立，如下：

```
void testBookEntryClass()
{
  BookEntry b("Addison-Wesley Publishing Company",
              "One Jacob Way, Reading, MA 01867");
...
}
```

并且在构造 **b** 的过程中，一个异常被抛出，**b** 的析构函数不会被调用。而且如果你试图采取主动手段处理异常情况，即当异常发生时调用 **delete**，如下所示：

```
void testBookEntryClass()
{
  BookEntry *pb = 0;
  try {
    pb = new BookEntry("Addison-Wesley Publishing Company",
                       "One Jacob Way, Reading, MA 01867");
    ...
  }
  catch (...) {                 // 捕获所有异常
     delete pb;                 // 删除 pb,当抛出异常时
     throw;                     // 传递异常给调用者
  }
  delete pb;                    // 正常删除 pb
}
```

你会发现在 **BookEntry** 构造函数里为 **Image** 分配的内存仍旧被丢失了，这是因为如果 **new** 操作没有成功完成，程序不会对 **pb** 进行赋值操作。如果 **BookEntry** 的构造函数抛出一

个异常，pb 将是一个空值，所以在 catch 块中删除它除了让你自己感觉良好以外没有任何作用。用灵巧指针（smart pointer）类 auto_ptr<BookEntry>（参见条款 M9）代替 raw BookEntry*也不会也什么作用，因为 new 操作成功完成前，也没有对 pb 进行赋值操作。

C++拒绝为没有完成构造操作的对象调用析构函数是有一些原因的，而不是故意为你制造困难。原因是：在很多情况下这么做是没有意义的，甚至是有害的。如果为没有完成构造操作的对象调用析构函数，析构函数如何去做呢？仅有的办法是在每个对象里加入一些字节来指示构造函数执行了多少步？然后让析构函数检测这些字节并判断该执行哪些操作。这样的记录会减慢析构函数的运行速度，并使得对象的尺寸变大。C++避免了这种开销，但是代价是不能自动地删除被部分构造的对象。（类似这种在程序行为与效率这间进行折衷处理的例子还可以参见 Effective C++条款 13）

因为当对象在构造中抛出异常后 C++不负责清除对象，所以你必须重新设计你的构造函数以让它们自己清除。经常用的方法是捕获所有的异常，然后执行一些清除代码，最后再重新抛出异常让它继续转递。如下所示，在 BookEntry 构造函数中使用这个方法：

```
BookEntry::BookEntry(const string& name,
                     const string& address,
                     const string& imageFileName,
                     const string& audioClipFileName)
: theName(name), theAddress(address),
  theImage(0), theAudioClip(0)
{
  try {                              // 这 try block 是新加入的
    if (imageFileName != "") {
      theImage = new Image(imageFileName);
    }
  if (audioClipFileName != "") {
      theAudioClip = new AudioClip(audioClipFileName);
    }
  }
  catch (...) {                      // 捕获所有异常
    delete theImage;                 // 完成必要的清除代码
    delete theAudioClip;
    throw;                           // 继续传递异常
  }
}
```

不用为 **BookEntry** 中的非指针数据成员操心，在类的构造函数被调用之前数据成员就被自动地初始化。所以如果 **BookEntry** 构造函数体开始执行，对象的 **theName, theAddress** 和 **thePhones** 数据成员已经被完全构造好了。这些数据可以被看做是完全构造的对象，所以它们将被自动释放，不用你介入操作。当然如果这些对象的构造函数调用可能会抛出异常的函数，那么那些构造函数必须自己去考虑捕获异常并在继续传递这些异常之前完成必需的清除操作。

你可能已经注意到 **BookEntry** 构造函数的 **catch** 块中的语句与在 **BookEntry** 的析构函数的语句几乎一样。这里的代码重复是绝对不可容忍的，所以最好的方法是把通用代码移入一个私有 **helper function** 中，让构造函数与析构函数都调用它。

```
class BookEntry {
public:
  ...                       // 同上
private:
  ...
  void cleanup();           // 通用清除代码
};
void BookEntry::cleanup()
{
  delete theImage;
  delete theAudioClip;
}
BookEntry::BookEntry(const string& name,
                     const string& address,
                      const string& imageFileName,
                       const string& audioClipFileName)
: theName(name), theAddress(address),
  theImage(0), theAudioClip(0)
{
  try {
    ...                     // 同上
  }
  catch (...)   {
    cleanup();              // 释放资源
    throw;                  // 传递异常
```

```
  }
}
BookEntry::~BookEntry()
{
  cleanup();
}
```

这似乎行了，但是它没有考虑到下面这种情况。假设我们略微改动一下设计，让 **theImage** 和 **theAudioClip** 是常量（**constant**）指针类型：

```
class BookEntry {
public:
  ...                                  // 同上
private:
  ...
  Image * const theImage;              // 指针现在是
  AudioClip * const theAudioClip;      // const 类型
};
```

必须通过 **BookEntry** 构造函数的成员初始化表来初始化这样的指针，因为再也没有其它地方可以给 **const** 指针赋值（参见 **Effective C++**条款 **12**）。通常会这样初始化 **theImage** 和 **theAudioClip**：

```
// 一个可能在异常抛出时导致资源泄漏的实现方法
BookEntry::BookEntry(const string& name,
                     const string& address,
                     const string& imageFileName,
                     const string& audioClipFileName)
: theName(name), theAddress(address),
  theImage(imageFileName != ""
        ? new Image(imageFileName)
        : 0),
  theAudioClip(audioClipFileName != ""
          ? new AudioClip(audioClipFileName)
          : 0)
{}
```

这样做导致我们原先一直想避免的问题重新出现：如果 **theAudioClip** 初始化时一个异常被抛出，**theImage** 所指的对象不会被释放。而且我们不能通过在构造函数中增加 **try** 和

catch 语句来解决问题，因为 **try** 和 **catch** 是语句，而成员初始化表仅允许有表达式（这也是为什么我们必须在 **theImage** 和 **theAudioClip** 的初始化中使用**?:**以代替 **if-then-else** 的原因）。

无论如何，在异常传递之前完成清除工作的唯一的方法就是捕获这些异常，所以如果我们不能在成员初始化表中放入 **try** 和 **catch** 语句，我们把它们移到其它地方。一种可能是在私有成员函数中，用这些函数返回指针指向初始化过的 **theImage** 和 **theAudioClip** 对象。

```
class BookEntry {
public:
  ...                       // 同上
private:
  ...                       // 数据成员同上
Image * initImage(const string& imageFileName);
  AudioClip * initAudioClip(const string&
                            audioClipFileName);
};
BookEntry::BookEntry(const string& name,
                     const string& address,
                     const string& imageFileName,
                     const string& audioClipFileName)
: theName(name), theAddress(address),
  theImage(initImage(imageFileName)),
  theAudioClip(initAudioClip(audioClipFileName))
{}
// theImage 被首先初始化,所以即使这个初始化失败也
// 不用担心资源泄漏，这个函数不用进行异常处理。
Image * BookEntry::initImage(const string& imageFileName)
{
  if (imageFileName != "") return new Image(imageFileName);
  else return 0;
}
// theAudioClip 被第二个初始化，所以如果在 theAudioClip
// 初始化过程中抛出异常，它必须确保 theImage 的资源被释放。
// 因此这个函数使用 try...catch 。
AudioClip * BookEntry::initAudioClip(const string&
```

```
                                        audioClipFileName)
{
  try {
    if (audioClipFileName != "") {
      return new AudioClip(audioClipFileName);
    }
    else return 0;
  }
  catch (...)   {
    delete theImage;
    throw;
  }
}
```

上面的程序的确不错，也解决了令我们头疼不已的问题。不过也有缺点，在原则上应该属于构造函数的代码却分散在几个函数里，这令我们很难维护。

更好的解决方法是采用条款 M9 的建议，把 **theImage** 和 **theAudioClip** 指向的对象做为一个资源，被一些局部对象管理。这个解决方法建立在这样一个事实基础上：**theImage** 和 **theAudioClip** 是两个指针，指向动态分配的对象，因此当指针消失的时候，这些对象应该被删除。**auto_ptr** 类就是基于这个目的而设计的。（参见条款 M9）因此我们把 **theImage** 和 **theAudioClip raw** 指针类型改成对应的 **auto_ptr** 类型。

```
class BookEntry {
public:
  ...                                       // 同上
private:
  ...
  const auto_ptr<Image> theImage;           // 它们现在是
  const auto_ptr<AudioClip> theAudioClip;   // auto_ptr 对象
};
```

这样做使得 **BookEntry** 的构造函数即使在存在异常的情况下也能做到不泄漏资源，而且让我们能够使用成员初始化表来初始化 **theImage** 和 **theAudioClip**，如下所示：

```
BookEntry::BookEntry(const string& name,
                     const string& address,
                     const string& imageFileName,
                     const string& audioClipFileName)
```

```
: theName(name), theAddress(address),
  theImage(imageFileName != ""
        ? new Image(imageFileName)
        : 0),
  theAudioClip(audioClipFileName != ""
          ? new AudioClip(audioClipFileName)
          : 0)
{}
```

在这里，如果在初始化 **theAudioClip** 时抛出异常，**theImage** 已经是一个被完全构造的对象，所以它能被自动删除掉，就象 **theName**，**theAddress** 和 **thePhones** 一样。而且因为 **theImage** 和 **theAudioClip** 现在是包含在 **BookEntry** 中的对象，当 **BookEntry** 被删除时它们能被自动地删除。因此不需要手工删除它们所指向的对象。可以这样简化 **BookEntry** 的析构函数：

```
BookEntry::~BookEntry()
{}                                        // nothing to do!
```

这表示你能完全去掉 **BookEntry** 的析构函数。

综上所述，如果你用对应的 **auto_ptr** 对象替代指针成员变量，就可以防止构造函数在存在异常时发生资源泄漏，你也不用手工在析构函数中释放资源，并且你还能象以前使用非 **const** 指针一样使用 **const** 指针，给其赋值。

在对象构造中，处理各种抛出异常的可能，是一个棘手的问题，但是 **auto_ptr(**或者类似于 **auto_ptr** 的类**)**能化繁为简。它不仅把令人不好理解的代码隐藏起来，而且使得程序在面对异常的情况下也能保持正常运行。

5.3 Item M11：禁止异常信息（exceptions）传递到析构函数外

在有两种情况下会调用析构函数。第一种是在正常情况下删除一个对象，例如对象超出了作用域或被显式地 **delete**。第二种是异常传递的堆栈辗转开解（**stack-unwinding**）过程中，由异常处理系统删除一个对象。

在上述两种情况下，调用析构函数时异常可能处于激活状态也可能没有处于激活状态。遗憾的是没有办法在析构函数内部区分出这两种情况。因此在写析构函数时你必须保守地假设有异常被激活。因为如果在一个异常被激活的同时，析构函数也抛出异常，并导致程序控制权转移到析构函数外，**C++**将调用 **terminate** 函数。这个函数的作用正如其名字所表示的：它终止你程序的运行，而且是立即终止，甚至连局部对象都没有被释放。

下面举一个例子，一个 **Session** 类用来跟踪在线计算机的 **sessions**，**session** 就是运行在从你一登录计算机开始一直到注销出系统为止的这段期间的某种东西。每个 **Session** 对象

关注的是它建立与释放的日期和时间：

```
class Session {
public:
  Session();
  ~Session();
  ...
private:
  static void logCreation(Session *objAddr);
  static void logDestruction(Session *objAddr);
};
```

函数 **logCreation** 和 **logDestruction** 被分别用于记录对象的建立与释放。我们因此可以这样编写 **Session** 的析构函数：

```
Session::~Session()
{
  logDestruction(this);
}
```

一切看上去很好，但是如果 **logDestruction** 抛出一个异常，会发生什么事呢？异常没有被 **Session** 的析构函数捕获住，所以它被传递到析构函数的调用者那里。但是如果析构函数本身的调用就是源自于某些其它异常的抛出，那么 **terminate** 函数将被自动调用，彻底终止你的程序。这不是你所希望发生的事情。程序没有记录下释放对象的信息，这是不幸的，甚至是一个大麻烦。那么事态果真严重到了必须终止程序运行的地步了么？如果没有，你必须防止在 **logDestruction** 内抛出的异常传递到 **Session** 析构函数的外面。唯一的方法是用 **try** 和 **catch blocks**。一种很自然的做法会这样编写函数：

```
Session::~Session()
{
  try {
    logDestruction(this);
  }
  catch (...) {
    cerr  << "Unable to log destruction of Session object "
          << "at address "
          << this
          << ".\n";
  }
```

```
}
```

但是这样做并不比你原来的代码安全。如果在 **catch** 中调用 **operator<<**时导致一个异常被抛出，我们就又遇到了老问题，一个异常被转递到 **Session** 析构函数的外面。

我们可以在 **catch** 中放入 **try**，但是这总得有一个限度，否则会陷入循环。因此我们在释放 **Session** 时必须忽略掉所有它抛出的异常：

```
Session::~Session()
{
  try {
    logDestruction(this);
  }
  catch (...) {  }
}
```

catch 表面上好像没有做任何事情，这是一个假象，实际上它阻止了任何从 **logDestruction** 抛出的异常被传递到 **session** 析构函数的外面。我们现在能高枕无忧了，无论 **session** 对象是不是在堆栈退栈（**stack unwinding**）中被释放，**terminate** 函数都不会被调用。

不允许异常传递到析构函数外面还有第二个原因。如果一个异常被析构函数抛出而没有在函数内部捕获住，那么析构函数就不会完全运行（它会停在抛出异常的那个地方上）。如果析构函数不完全运行，它就无法完成希望它做的所有事情。例如，我们对 **session** 类做一个修改，在建立 **session** 时启动一个数据库事务（**database transaction**）,终止 **session** 时结束这个事务：

```
Session::Session()          // 为了简单起见，,
{                           // 这个构造函数没有
                            // 处理异常
  logCreation(this);
  startTransaction();       // 启动 database transaction
}
Session::~Session()
{
  logDestruction(this);
  endTransaction();         // 结束 database transaction
}
```

如果在这里 **logDestruction** 抛出一个异常，在 **session** 构造函数内启动的 **transaction** 就没有被终止。我们也许能够通过重新调整 **session** 析构函数内的函数调用顺序来消除问

题，但是如果 **endTransaction** 也抛出一个异常，我们除了回到使用 **try** 和 **catch** 外，别无选择。

综上所述，我们知道禁止异常传递到析构函数外有两个原因，第一能够在异常转递的堆栈辗转开解（**stack-unwinding**）的过程中，防止 **terminate** 被调用。第二它能帮助确保析构函数总能完成我们希望它做的所有事情。（如果你仍旧不很信服我所说的理由，可以去看 **Herb Sutter** 的文章 **Exception-Safe Generic Containers**，特别是“**Destructors That Throw and Why They’re Evil**”这段）。

5.4 Item M12：理解“抛出一个异常”与“传递一个参数”或“调用一个虚函数”间的差异

从语法上看，在函数里声明参数与在 **catch** 子句中声明参数几乎没有什么差别：

```
class Widget { ... };                    //一个类，具体是什么类
                                         // 在这里并不重要
void f1(Widget w);                       // 一些函数，其参数分别为
void f2(Widget& w);                      // Widget, Widget&,或
void f3(const Widget& w);                // Widget* 类型
void f4(Widget *pw);
void f5(const Widget *pw);
catch (Widget w) ...                     //一些 catch 子句，用来
catch (Widget& w)   ...                  //捕获异常，异常的类型为
catch (const Widget& w) ...              // Widget, Widget&, 或
catch (Widget *pw) ...                   // Widget*
catch (const Widget *pw) ...
```

你因此可能会认为用 **throw** 抛出一个异常到 **catch** 子句中与通过函数调用传递一个参数两者基本相同。这里面确有一些相同点，但是他们也存在着巨大的差异。

让我们先从相同点谈起。你传递函数参数与异常的途径可以是传值、传递引用或传递指针，这是相同的。但是当你传递参数和异常时，系统所要完成的操作过程则是完全不同的。产生这个差异的原因是：你调用函数时，程序的控制权最终还会返回到函数的调用处，但是当你抛出一个异常时，控制权永远不会回到抛出异常的地方。

有这样一个函数，参数类型是 **Widget**，并抛出一个 **Widget** 类型的异常：

```
// 一个函数，从流中读值到 Widget 中
istream operator>>(istream& s, Widget& w);
void passAndThrowWidget()
{
  Widget localWidget;
```

```
  cin >> localWidget;          //传递 localWidget 到 operator>>
  throw localWidget;           // 抛出 localWidget 异常
}
```

当传递 localWidget 到函数 operator>>里，不用进行拷贝操作，而是把 operator>>内的引用类型变量 w 指向 localWidget，任何对 w 的操作实际上都施加到 localWidget 上。这与抛出 localWidget 异常有很大不同。不论通过传值捕获异常还是通过引用捕获（不能通过指针捕获这个异常，因为类型不匹配）都将进行 lcalWidget 的拷贝操作，也就说传递到 catch 子句中的是 localWidget 的拷贝。必须这么做，因为当 localWidget 离开了生存空间后，其析构函数将被调用。如果把 localWidget 本身（而不是它的拷贝）传递给 catch 子句，这个子句接收到的只是一个被析构了的 Widget，一个 Widget 的“尸体”。这是无法使用的。因此 C++规范要求被做为异常抛出的对象必须被复制。

即使被抛出的对象不会被释放，也会进行拷贝操作。例如如果 passAndThrowWidget 函数声明 localWidget 为静态变量（static），

```
void passAndThrowWidget()
{
  static Widget localWidget;        // 现在是静态变量（static）;
                                    //一直存在至程序结束
  cin >> localWidget;               // 象以前那样运行
  throw localWidget;                // 仍将对 localWidget
}                                   //进行拷贝操作
```

当抛出异常时仍将复制出 localWidget 的一个拷贝。这表示即使通过引用来捕获异常，也不能在 catch 块中修改 localWidget；仅仅能修改 localWidget 的拷贝。对异常对象进行强制复制拷贝，这个限制有助于我们理解参数传递与抛出异常的第二个差异：抛出异常运行速度比参数传递要慢。

当异常对象被拷贝时，拷贝操作是由对象的拷贝构造函数完成的。该拷贝构造函数是对象的静态类型（static type）所对应类的拷贝构造函数，而不是对象的动态类型（dynamic type）对应类的拷贝构造函数。比如以下这经过少许修改的 passAndThrowWidget：

```
class Widget { ... };
class SpecialWidget: public Widget { ... };
void passAndThrowWidget()
{
  SpecialWidget localSpecialWidget;
  ...
  Widget& rw = localSpecialWidget;      // rw 引用 SpecialWidget
```

```
  throw rw;                          //它抛出一个类型为 Widget
                                     // 的异常
}
```

这里抛出的异常对象是 **Widget**，即使 **rw** 引用的是一个 **SpecialWidget**。因为 **rw** 的静态类型（**static type**）是 **Widget**，而不是 **SpecialWidget**。你的编译器根本没有注意到 **rw** 引用的是一个 **SpecialWidget**。编译器所注意的是 **rw** 的静态类型（**static type**）。这种行为可能与你所期待的不一样，但是这与在其他情况下 **C++**中拷贝构造函数的行为是一致的。（不过有一种技术可以让你根据对象的动态类型 **dynamic type** 进行拷贝，参见条款 **M25**）

异常是其它对象的拷贝，这个事实影响到你如何在 **catch** 块中再抛出一个异常。比如下面这两个 **catch** 块，乍一看好像一样：

```
catch (Widget& w)                    // 捕获 Widget 异常
{
  ...                                // 处理异常
  throw;                             // 重新抛出异常，让它
}                                    // 继续传递
catch (Widget& w)                    // 捕获 Widget 异常
{
  ...                                // 处理异常
  throw w;                           // 传递被捕获异常的
}                                    // 拷贝
```

这两个 **catch** 块的差别在于第一个 **catch** 块中重新抛出的是当前捕获的异常，而第二个 **catch** 块中重新抛出的是当前捕获异常的一个新的拷贝。如果忽略生成额外拷贝的系统开销，这两种方法还有差异么？

当然有。第一个块中重新抛出的是当前异常（**current exception**）,无论它是什么类型。特别是如果这个异常开始就是做为 **SpecialWidget** 类型抛出的，那么第一个块中传递出去的还是 **SpecialWidget** 异常，即使 **w** 的静态类型（**static type**）是 **Widget**。这是因为重新抛出异常时没有进行拷贝操作。第二个 **catch** 块重新抛出的是新异常，类型总是 **Widget**，因为 **w** 的静态类型（**static type**）是 **Widget**。一般来说，你应该用 **throw** 来重新抛出当前的异常，因为这样不会改变被传递出去的异常类型，而且更有效率，因为不用生成一个新拷贝。（顺便说一句，异常生成的拷贝是一个临时对象。正如条款 **19** 解释的，临时对象能让编译器优化它的生存期（**optimize it out of existence**），不过我想你的编译器很难这么做，因为程序中很少发生异常，所以编译器厂商不会在这方面花大量的精力。）

让我们测试一下下面这三种用来捕获 **Widget** 异常的 **catch** 子句，异常是做为

passAndThrowWidgetp 抛出的：

```
catch (Widget w) ...                    // 通过传值捕获异常
catch (Widget& w) ...                   // 通过传递引用捕获
                                        // 异常
catch (const Widget& w) ...             //通过传递指向 const 的引用
                                        //捕获异常
```

我们立刻注意到了传递参数与传递异常的另一个差异。一个被异常抛出的对象（刚才解释过，总是一个临时对象）可以通过普通的引用捕获；它不需要通过指向 const 对象的引用（reference-to-const）捕获。在函数调用中不允许转递一个临时对象到一个非 const 引用类型的参数里（参见条款 M19），但是在异常中却被允许。

让我们先不管这个差异，回到异常对象拷贝的测试上来。我们知道当用传值的方式传递函数的参数，我们制造了被传递对象的一个拷贝（参见 Effective C++ 条款 22），并把这个拷贝存储到函数的参数里。同样我们通过传值的方式传递一个异常时，也是这么做的。当我们这样声明一个 catch 子句时：

```
catch (Widget w) ...                    // 通过传值捕获
```

会建立两个被抛出对象的拷贝，一个是所有异常都必须建立的临时对象，第二个是把临时对象拷贝进 w 中（**WQ 加注，重要：是两个!**）。同样，当我们通过引用捕获异常时，

```
catch (Widget& w) ...                   // 通过引用捕获
catch (const Widget& w) ...             //也通过引用捕获
```

这仍旧会建立一个被抛出对象的拷贝：拷贝同样是一个临时对象。相反当我们通过引用传递函数参数时，没有进行对象拷贝。当抛出一个异常时，系统构造的（以后会析构掉）被抛出对象的拷贝数比以相同对象做为参数传递给函数时构造的拷贝数要多一个。

我们还没有讨论通过指针抛出异常的情况。不过，通过指针抛出异常与通过指针传递参数是相同的。不论哪种方法都是一个指针的拷贝被传递。但，你不能认为抛出的指针是一个指向局部对象的指针，因为当异常离开局部变量的生存空间时，该局部变量已经被释放。Catch 子句将获得一个指向已经不存在的对象的指针。这种行为在设计时应该予以避免。**（WQ 加注，也就是说：必须是全局的或堆中的。）**

对象从函数的调用处传递到函数参数里与从异常抛出点传递到 catch 子句里所采用的方法不同，这只是参数传递与异常传递的区别的一个方面；第二个差异是在函数调用者或抛出异常者与被调用者或异常捕获者之间的类型匹配的过程不同。比如在标准数学库（the standard math library）中 sqrt 函数：

```
double sqrt(double);                    // from <cmath> or <math.h>
```

我们能这样计算一个整数的平方根，如下所示：

```
int i;
```

```
double sqrtOfi = sqrt(i);
```

毫无疑问，C++允许进行从 int 到 double 的隐式类型转换，所以在 sqrt 的调用中，i 被悄悄地转变为 double 类型，并且其返回值也是 double。（有关隐式类型转换的详细讨论参见条款 M5）一般来说，catch 子句匹配异常类型时不会进行这样的转换。见下面的代码：

```
void f(int value)
{
  try {
    if (someFunction()) {      // 如果 someFunction()返回
      throw value;             //真，抛出一个整形值
    ...
    }
  }
  catch (double d) {           // 只处理 double 类型的异常
    ...
  }
  ...
}
```

在 try 块中抛出的 int 异常不会被处理 double 异常的 catch 子句捕获。该子句只能捕获类型真真正正为 double 的异常，不进行类型转换。因此如果要想捕获 int 异常，必须使用带有 int 或 int&参数的 catch 子句。

不过在 catch 子句中进行异常匹配时可以进行两种类型转换。第一种是继承类与基类间的转换。一个用来捕获基类的 catch 子句也可以处理派生类类型的异常。例如在标准 C++库（STL）定义的异常类层次中的诊断部分（diagnostics portion ）(参见 Effective C++ 条款 49)。

捕获 runtime_errors 异常的 Catch 子句可以捕获 range_error 类型和 overflow_error 类型的异常；可以接收根类 exception 异常的 catch 子句能捕获其任意派生类异常。

这种派生类与基类（inheritance_based）间的异常类型转换可以作用于数值、引用以及指针上：

```
catch (runtime_error) ...              // can catch errors of type
catch (runtime_error&) ...             // runtime_error,
catch (const runtime_error&) ...       // range_error, or
                                  // overflow_error
catch (runtime_error*) ...             // can catch errors of type
catch (const runtime_error*) ...       // runtime_error*,
```

```
                                    // range_error*, or
                                    // overflow_error*
```

第二种是允许从一个类型化指针（typed pointer）转变成无类型指针（untyped pointer），所以带有 const void* 指针的 catch 子句能捕获任何类型的指针类型异常：

```
catch (const void*) ...             //捕获任何指针类型异常
```

传递参数和传递异常间最后一点差别是 catch 子句匹配顺序总是取决于它们在程序中出现的顺序。因此一个派生类异常可能被处理其基类异常的 catch 子句捕获，即使同时存在有能直接处理该派生类异常的 catch 子句，与相同的 try 块相对应。例如：

```
try {
  ...
}
catch (logic_error& ex) {           // 这个 catch 块 将捕获
  ...                               // 所有的 logic_error
}                                   // 异常，包括它的派生类

catch (invalid_argument& ex) {      // 这个块永远不会被执行
  ...                               //因为所有的
}                                   // invalid_argument
                                    // 异常 都被上面的
                                    // catch 子句捕获。
```

与上面这种行为相反，当你调用一个虚拟函数时，被调用的函数位于与发出函数调用的对象的动态类型（dynamic type）最相近的类里。你可以这样说虚拟函数采用最优适合法，而异常处理采用的是最先适合法。如果一个处理派生类异常的 catch 子句位于处理基类异常的 catch 子句后面，编译器会发出警告。（因为这样的代码在 C++里通常是不合法的。）不过你最好做好预先防范：不要把处理基类异常的 catch 子句放在处理派生类异常的 catch 子句的前面。象上面那个例子，应该这样去写：

```
try {
  ...
}
catch (invalid_argument& ex) {      // 处理 invalid_argument
  ...                               //异常
}
catch (logic_error& ex) {           // 处理所有其它的
  ...                               // logic_errors 异常
```

```
}
```

综上所述，把一个对象传递给函数或一个对象调用虚拟函数与把一个对象做为异常抛出，这之间有三个主要区别。第一、异常对象在传递时总被进行拷贝；当通过传值方式捕获时，异常对象被拷贝了两次。对象做为参数传递给函数时不一定需要被拷贝。第二、对象做为异常被抛出与做为参数传递给函数相比，前者类型转换比后者要少（前者只有两种转换形式）。最后一点，**catch** 子句进行异常类型匹配的顺序是它们在源代码中出现的顺序，第一个类型匹配成功的 **catch** 将被用来执行。当一个对象调用一个虚拟函数时，被选择的函数位于与对象类型匹配最佳的类里，即使该类不是在源代码的最前头。

5.5 Item M13：通过引用（reference）捕获异常

当你写一个 **catch** 子句时，必须确定让异常通过何种方式传递到 **catch** 子句里。你可以有三个选择：与你给函数传递参数一样，通过指针（**by pointer**），通过传值（**by value**）或通过引用（**by reference**）。

我们首先讨论通过指针方式捕获异常（**catch by pointer**）。从 **throw** 处传递一个异常到 **catch** 子句是一个缓慢的过程，在理论上这种方法的实现对于这个过程来说是效率最高的。因为在传递异常信息时，只有采用通过指针抛出异常的方法才能够做到不拷贝对象（参见条款 **M12**），例如：

```
class exception { ... };          // 来自标准 C++库（STL）
                                  // 中的异常类层次
                                  // （参见条款 12）
void someFunction()
{
  static exception ex;            // 异常对象
  ...
  throw &ex;                      // 抛出一个指针，指向 ex
  ...
}
void doSomething()
{
  try {
    someFunction();               // 抛出一个 exception*
  }
  catch (exception *ex) {         // 捕获 exception*;
    ...                           // 没有对象被拷贝
```

```
  }
}
```

这看上去很不错，但是实际情况却不是这样。为了能让程序正常运行，程序员定义异常对象时必须确保当程序控制权离开抛出指针的函数后，对象还能够继续生存。全局与静态对象都能够做到这一点，但是程序员很容易忘记这个约束。如果真是如此的话，他们会这样写代码：

```
void someFunction()
{
  exception ex;                    // 局部异常对象;
                                   // 当退出函数的生存空间时
                                   // 这个对象将被释放。
  ...
  throw &ex;                       // 抛出一个指针，指向
  ...                              // 已被释放的对象
}
```

这简直糟糕透了，因为处理这个异常的 **catch** 子句接受到的指针，其指向的对象已经不再存在。

另一种抛出指针的方法是建立一个堆对象（**new heap object**）：

```
void someFunction()
{
  ...
  throw new exception;             // 抛出一个指针，指向一个在堆中
  ...                              // 建立的对象(希望
}                                  // 操作符 new — 参见条款 M8—
                                   // 自己不要再抛出一个
                                   // 异常!)
```

这避免了捕获一个指向已被释放对象的指针的问题，但是 **catch** 子句的作者又面临一个令人头疼的问题：他们是否应该删除他们接受的指针？如果是在堆中建立的异常对象，那他们必须删除它，否则会造成资源泄漏。如果不是在堆中建立的异常对象，他们绝对不能删除它，否则程序的行为将不可预测。该如何做呢？

这是不可能知道的。一些被调用者可能会传递全局或静态对象的地址，另一些可能传递堆中建立的异常对象的地址。通过指针捕获异常，将遇到一个哈姆雷特式的难题：是删除还是不删除？这是一个难以回答的问题。所以你最好避开它。

而且，通过指针捕获异常也不符合 **C++**语言本身的规范。四个标准的异常——

bad_alloc(当 operator new(参见条款 M8)不能分配足够的内存时，被抛出)，bad_cast（当 dynamic_cast 针对一个引用（reference）操作失败时，被抛出）,bad_typeid（当 dynamic_cast 对空指针进行操作时，被抛出）和 bad_exception（用于 unexpected 异常；参见条款 M14）——都不是指向对象的指针，所以你必须通过值或引用来捕获它们。

通过值捕获异常（catch-by-value）可以解决上述的问题，例如异常对象删除的问题和使用标准异常类型的问题。但是当它们被抛出时系统将对异常对象拷贝两次（参见条款 M12）。而且它会产生 slicing problem，即派生类的异常对象被做为基类异常对象捕获时，那它的派生类行为就被切掉了（sliced off）。这样的 sliced 对象实际上是一个基类对象：它们没有派生类的数据成员，而且当本准备调用它们的虚拟函数时，系统解析后调用的却是基类对象的函数。（当一个对象通过传值方式传递给函数，也会发生一样的情况——参见 Effective C++ 条款 22）。例如下面这个程序采用了扩展自标准异常类的异常类层次体系：

```
class exception {                    // 如上，这是
public:                              // 一个标准异常类
  virtual const char * what() throw();
                                     // 返回异常的简短描述.
  ...                                // （在函数声明的结尾处
                                     // 的"throw()"，
};                                   //有关它的信息
                                     // 参见条款 14)

class runtime_error:                 //也来自标准 C++异常类
  public exception { ... };
class Validation_error:              // 客户自己加入个类
  public runtime_error {
public:
  virtual const char * what() throw();
                                     // 重新定义在异常类中
  ...                                //虚拟函数
};                                   //
void someFunction()                  // 抛出一个 validation
{                                    // 异常
  ...
  if (a validation 测试失败) {
    throw Validation_error();
```

```
  }
...
}
void doSomething()
{
  try {
    someFunction();                 // 抛出 validation
  }                                 //异常
  catch (exception ex) {            //捕获所有标准异常类
                                    // 或它的派生类
  cerr << ex.what();                // 调用 exception::what(),
  ...                               // 而不是 Validation_error::what()
}
}
```

调用的是基类的 **what** 函数，即使被抛出的异常对象是 **runtime_error** 或 **Validation_error** 类型并且它们已经重新定义了这个虚拟函数。这种 **slicing** 行为绝不是你所期望的。

最后剩下方法就是通过引用捕获异常（**catch-by-reference**）。通过引用捕获异常能使你避开上述所有问题。不象通过指针捕获异常，这种方法不会有对象删除的问题而且也能捕获标准异常类型。也不象通过值捕获异常，这种方法没有 **slicing problem**，而且异常对象只被拷贝一次。

我们采用通过引用捕获异常的方法重写最后那个例子，如下所示：

```
void someFunction()                 //这个函数没有改变
{
  ...
  if (a validation 测试失败) {
    throw Validation_error();
  }
  ...
}
void doSomething()
{
  try {
    someFunction();                 // 没有改变
```

```
  }
  catch (exception& ex) {          // 这里，我们通过引用捕获异常
                                   // 以替代原来的通过值捕获
    cerr << ex.what();             // 现在调用的是
                                   // Validation_error::what(),
    ...                            // 而不是 exception::what()
  }
}
```

这里没有对 **throw** 进行任何改变，仅仅改变了 **catch** 子句，给它加了一个**&**符号。然而这个微小的改变能造成了巨大的变化，因为 **catch** 块中的虚拟函数能够如我们所愿那样工作了：调用的 **Validation_erro** 函数是我们重新定义过的函数。

如果你通过引用捕获异常（**catch by reference**），你就能避开上述所有问题，不会为是否删除异常对象而烦恼；能够避开 **slicing** 异常对象；能够捕获标准异常类型；减少异常对象需要被拷贝的数目。所以你还在等什么？通过引用捕获异常吧（**Catch exceptions by reference**）!

5.6 Item M14：审慎使用异常规格(exception specifications)

毫无疑问，异常规格是一个引人注目的特性。它使得代码更容易理解，因为它明确地描述了一个函数可以抛出什么样的异常。但是它不只是一个有趣的注释。编译器在编译时有时能够检测到异常规格的不一致。而且如果一个函数抛出一个不在异常规格范围里的异常，系统在运行时能够检测出这个错误，然后一个特殊函数 **unexpected** 将被自动地调用。异常规格既可以做为一个指导性文档同时也是异常使用的强制约束机制，它好像有着很诱人的外表。

不过在通常情况下，美貌只是一层皮，外表的美丽并不代表其内在的素质。函数 **unexpected** 缺省的行为是调用函数 **terminate**，而 **terminate** 缺省的行为是调用函数 **abort**，所以一个违反异常规格的程序其缺省的行为就是 **halt**（停止运行）。在激活的栈中的局部变量没有被释放，因为 **abort** 在关闭程序时不进行这样的清除操作。对异常规格的触犯变成了一场并不应该发生的灾难。

不幸的是，我们很容易就能够编写出导致发生这种灾难的函数。编译器仅仅部分地检测异常的使用是否与异常规格保持一致。一个函数调用了另一个函数，并且后者可能抛出一个违反前者异常规格的异常，（**A** 函数调用 **B** 函数，但因为 **B** 函数可能抛出一个不在 **A** 函数异常规格之内的异常，所以这个函数调用就违反了 **A** 函数的异常规格　译者注）编译器不对此种情况进行检测，并且语言标准也禁止编译器拒绝这种调用方式（尽管可以显示警告信息）。

例如函数 **f1** 没有声明异常规格，这样的函数就可以抛出任意种类的异常：

```
extern void f1();                    // 可以抛出任意的异常
```

假设有一个函数 **f2** 通过它的异常规格来声明其只能抛出 **int** 类型的异常：

```
void f2() throw(int);
```

f2 调用 **f1** 是非常合法的，即使 **f1** 可能抛出一个违反 **f2** 异常规格的异常：

```
void f2() throw(int)
{
  ...
  f1();                    // 即使 f1 可能抛出不是 int 类型的
                           //异常，这也是合法的。
  ...
}
```

当带有异常规格的新代码与没有异常规格的老代码整合在一起工作时，这种灵活性就显得很重要。

因为你的编译器允许你调用一个函数，其抛出的异常与发出调用的函数的异常规格不一致，并且这样的调用可能导致你的程序执行被终止，所以在编写软件时应该采取措施把这种不一致减小到最少。一种好方法是避免在带有类型参数的模板内使用异常规格。例如下面这种模板，它好像不能抛出任何异常：

```
// a poorly designed template wrt exception specifications
template<class T>
bool operator==(const T& lhs, const T& rhs) throw()
{
  return &lhs == &rhs;
}
```

这个模板为所有类型定义了一个操作符函数 **operator==**。对于任意一对类型相同的对象，如果对象有一样的地址，该函数返回 **true**，否则返回 **false**。

这个模板包含的异常规格表示模板生成的函数不能抛出异常。但是事实可能不会这样，因为 **opertor&(**地址操作符**,**参见 **Effective C++** 条款 **45)**能被一些类型对象重载。如果被重载的话，当调用从 **operator==**函数内部调用 **opertor&**时，**opertor&**可能会抛出一个异常，这样就违反了我们的异常规格，使得程序控制跳转到 **unexpected**。

上述的例子是一种更一般问题的特例，这个更一般问题也就是没有办法知道某种模板类型参数抛出什么样的异常。我们几乎不可能为一个模板提供一个有意义的异常规格。因为模板总是采用不同的方法使用类型参数。解决方法只能是模板和异常规格不要混合使用。

能够避免调用 **unexpected** 函数的第二个方法是如果在一个函数内调用其它没有异常规格的函数时应该去除这个函数的异常规格。这很容易理解，但是实际中容易被忽略。比如允

许用户注册一个回调函数：

```
// 一个 window 系统回调函数指针
//当一个 window 系统事件发生时
typedef void (*CallBackPtr)(int eventXLocation,
                            int eventYLocation,
                            void *dataToPassBack);
//window 系统类，含有回调函数指针，
//该回调函数能被 window 系统客户注册
class CallBack {
public:
  CallBack(CallBackPtr fPtr, void *dataToPassBack)
  : func(fPtr), data(dataToPassBack) {}
  void makeCallBack(int eventXLocation,
                    int eventYLocation) const throw();
private:
  CallBackPtr func;              // function to call when
                                 // callback is made
   void *data;                   // data to pass to callback
};                               // function
// 为了实现回调函数，我们调用注册函数，
//事件的作标与注册数据做为函数参数。
void CallBack::makeCallBack(int eventXLocation,
                            int eventYLocation) const throw()
{
  func(eventXLocation, eventYLocation, data);
}
```

这里在 makeCallBack 内调用 **func**，要冒违反异常规格的风险，因为无法知道 **func** 会抛出什么类型的异常。

通过在程序在 **CallBackPtr typedef** 中采用更严格的异常规格来解决问题：

```
typedef void (*CallBackPtr)(int eventXLocation,
                            int eventYLocation,
                            void *dataToPassBack) throw();
```

这样定义 **typedef** 后，如果注册一个可能会抛出异常的 **callback** 函数将是非法的：

```
// 一个没有异常规格的回调函数
```

```
void callBackFcn1(int eventXLocation, int eventYLocation,
                  void *dataToPassBack);
void *callBackData;
...
CallBack c1(callBackFcn1, callBackData);
                                  //错误！callBackFcn1 可能
                                  // 抛出异常
//带有异常规格的回调函数
void callBackFcn2(int eventXLocation,
                  int eventYLocation,
                  void *dataToPassBack) throw();
CallBack c2(callBackFcn2, callBackData);
                                  // 正确，callBackFcn2
                                  // 没有异常规格
```

传递函数指针时进行这种异常规格的检查，是语言的较新的特性，所以有可能你的编译器不支持这个特性。如果它们不支持，那就依靠你自己来确保不能犯这种错误。

避免调用 **unexpected** 的第三个方法是处理系统本身抛出的异常。这些异常中最常见的是 **bad_alloc**，当内存分配失败时它被 **operator new** 和 **operator new[]**抛出（参见条款 **M8**）。如果你在函数里使用 **new** 操作符（还参见条款 **M8**），你必须为函数可能遇到 **bad_alloc** 异常作好准备。

现在常说预防胜于治疗（即：做任何事都要未雨绸缪 译者注），但是有时却是预防困难而治疗容易。也就是说有时直接处理 **unexpected** 异常比防止它们被抛出要简单。例如你正在编写一个软件，精确地使用了异常规格，但是你必须从没有使用异常规格的程序库中调用函数，要防止抛出 **unexpected** 异常是不现实的，因为这需要改变程序库中的代码。

虽然防止抛出 **unexpected** 异常是不现实的，但是 **C++**允许你用其它不同的异常类型替换 **unexpected** 异常，你能够利用这个特性。例如你希望所有的 **unexpected** 异常都被替换为 **UnexpectedException** 对象。你能这样编写代码：

```
class UnexpectedException {};          // 所有的 unexpected 异常对象被
                                       //替换为这种类型对象
void convertUnexpected()               // 如果一个 unexpected 异常被
{                                      // 抛出，这个函数被调用
  throw UnexpectedException();
}
```

通过用 **convertUnexpected** 函数替换缺省的 **unexpected** 函数，来使上述代码开始运行。：

```
set_unexpected(convertUnexpected);
```

当你这么做了以后，一个 **unexpected** 异常将触发调用 **convertUnexpected** 函数。**Unexpected** 异常被一种 **UnexpectedException** 新异常类型替换。如果被违反的异常规格包含 **UnexpectedException** 异常，那么异常传递将继续下去，好像异常规格总是得到满足。（如果异常规格没有包含 **UnexpectedException**，**terminate** 将被调用，就好像你没有替换 **unexpected** 一样）

另一种把 **unexpected** 异常转变成知名类型的方法是替换 **unexpected** 函数，让其重新抛出当前异常，这样异常将被替换为 **bad_exception**。你可以这样编写：

```
void convertUnexpected()          // 如果一个 unexpected 异常被
{                                 //抛出，这个函数被调用
  throw;                          // 它只是重新抛出当前
}                                 // 异常
set_unexpected(convertUnexpected);
                                  // 安装 convertUnexpected
                                  // 做为 unexpected
                                  // 的替代品
```

如果这么做，你应该在所有的异常规格里包含 **bad_exception**（或它的基类，标准类 **exception**）。你将不必再担心如果遇到 **unexpected** 异常会导致程序运行终止。任何不听话的异常都将被替换为 **bad_exception**，这个异常代替原来的异常继续传递。

到现在你应该理解异常规格能导致大量的麻烦。编译器仅仅能部分地检测它们的使用是否一致，在模板中使用它们会有问题，一不注意它们就很容易被违反，并且在缺省的情况下它们被违反时会导致程序终止运行。异常规格还有一个缺点就是它们能导致 **unexpected** 被触发，即使一个 **high-level** 调用者准备处理被抛出的异常，比如下面这个几乎一字不差地来自从条款 **M11** 例子：

```
class Session {                   // for modeling online
public:                           // sessions
  ~Session();
  ...
private:
  static void logDestruction(Session *objAddr) throw();
};
Session::~Session()
{
  try {
```

```
    logDestruction(this);
  }
  catch (...) {  }
}
```

session 的析构函数调用 logDestruction 记录有关 session 对象被释放的信息，它明确地要捕获从 logDestruction 抛出的所有异常。但是 logDestruction 的异常规格表示其不抛出任何异常。现在假设被 logDestruction 调用的函数抛出了一个异常，而 logDestruction 没有捕获。我们不会期望发生这样的事情，但正如我们所见，很容易就会写出违反异常规格的代码。当这个异常通过 logDestruction 传递出来，unexpected 将被调用，缺省情况下将导致程序终止执行。这是一个正确的行为，但这是 session 析构函数的作者所希望的行为么？作者想处理所有可能的异常，所以好像不应该不给 session 析构函数里的 catch 块执行的机会就终止程序。如果 logDestruction 没有异常规格，这种事情就不会发生（一种防止的方法是如上所描述的那样替换 unexpected）。

以全面的角度去看待异常规格是非常重要的。它们提供了优秀的文档来说明一个函数抛出异常的种类，并且在违反它的情况下，会有可怕的结果，程序被立即终止，在缺省时它们会这么做。同时编译器只会部分地检测它们的一致性，所以他们很容易被不经意地违反。而且他们会阻止 high-level 异常处理器来处理 unexpected 异常，即使这些异常处理器知道如何去做。

综上所述，异常规格是一个应被审慎使用的特性。在把它们加入到你的函数之前，应考虑它们所带来的行为是否就是你所希望的行为。

5.7 Item M15：了解异常处理的系统开销

为了在运行时处理异常，程序要记录大量的信息。无论执行到什么地方，程序都必须能够识别出如果在此处抛出异常的话，将要被释放哪一个对象；程序必须知道每一个入口点，以便从 try 块中退出；对于每一个 try 块，他们都必须跟踪与其相关的 catch 子句以及这些 catch 子句能够捕获的异常类型。这种信息的记录不是没有代价的。虽然确保程序满足异常规格不需要运行时的比较（runtime comparisons），而且当异常被抛出时也不用额外的开销来释放相关的对象和匹配正确的 catch 字句。但是异常处理确是有代价的，即使你没有使用 try，throw 或 catch 关键字，你同样得付出一些代价。

让我们先从你不使用任何异常处理特性也要付出的代价谈起。你需要空间建立数据结构来跟踪对象是否被完全构造（constructed）(参见条款 M10)，你也需要 CPU 时间保持这些数据结构不断更新。这些开销一般不是很大，但是采用不支持异常的方法编译的程序一般比支持异常的程序运行速度更快所占空间也更小。

在理论上，你不能对这些代价进行选择：异常是 C++的一部分，C++编译器必须支持异

常。也就是说，当你不用异常处理时你不能让编译器生产商消除这方面的开销，因为程序一般由多个独立生成的目标文件（**object files**）组成，只有一个目标文件不进行异常处理并不能代表其他目标文件不进行异常处理。而且即使组成可执行文件的目标文件都不进行异常处理，那么还有它们所连接的程序库呢？如果程序的任何一部分使用了异常，其它部分必须也支持异常。否则在运行时程序就不可能提供正确的异常处理。

不过这只是理论，实际上大部分支持异常的编译器生产商都允许你自由控制是否在生成的代码里包含进支持异常的内容。如果你知道你程序的任何部分都不使用 **try**，**throw** 或 **catch**，并且你也知道所连接的程序库也没有使用 **try**，**throw** 或 **catch**，你就可以采用不支持异常处理的方法进行编译，这可以缩小程序的尺寸和提高速度，否则你就得为一个不需要的特性而付出代价。随着时间的推移，使用异处理的程序库开始变得普遍了，上面这种方法将逐渐不能使用，但是根据目前的软件开发情况来看，如果你已经决定不使用任何的异常特性，那么采用不支持异常的方法编译程序是一个性能优化的合理方法。同样这对于想避开异常的程序库来说也是一个性能优化的好方法，这能保证异常不会从客户端程序传递进程序库里，不过同时这样做也会妨碍客户端程序重定义程序库中声明的虚拟函数，并且不允许有在客户端定义的回调函数。

使用异常处理的第二个开销来自于 **try** 块，无论何时使用它，也就是当你想能够捕获异常时，那你都得为此付出代价。不同的编译器实现 **try** 块的方法不同，所以编译器与编译器间的开销也不一样。粗略地估计，如果你使用 **try** 块，代码的尺寸将增加 **5**％－**10**％并且运行速度也同比例减慢。这还是假设程序没有抛出异常，我这里讨论的只是在程序里使用 **try** 块的开销。为了减少开销，你应该避免使用无用的 **try** 块。

编译器为异常规格生成的代码与它们为 **try** 块生成的代码一样多，所以一个异常规格一般花掉与 **try** 块一样多的系统开销。什么？你说你认为异常规格只是一个规格而已，你认为它们不会产生代码？那么好，现在你应该对此有新的认识了。

现在我们来到了问题的核心部分，看看抛出异常的开销。事实上我们不用太关心这个问题，因为异常是很少见的，这种事件的发生往往被描述为 **exceptional**（异常的，罕见的）。**80**－**20** 规则（参见条款 **M16**）告诉我们这样的事件不会对整个程序的性能造成太大的影响。但是我知道你仍旧好奇地想知道如果抛出一个异常到底会有多大的开销，答案是这可能会比较大。与一个正常的函数返回相比，通过抛出异常从函数里返回可能会慢三个数量级。这个开销很大。但是仅仅当你抛出异常时才会有这个开销，一般不会发生。但是如果你用异常表示一个比较普遍的状况，例如完成对数据结构的遍历或结束一个循环，那你必须重新予以考虑。

不过请等一下，你问我是怎么知道这些事情的呢？如果说支持异常对于大多数编译器来说是一个较新的特性，如果说不同的编译器的异常方法也不同，那么我如何能说程序的尺寸将增大 **5%-10%**，它的速度也同比例减慢，而且如果有大量的异常被抛出，程序运行速度会

呈数量级的减慢呢？答案是令人惊恐的：一些传闻和一些基准测试（benchmarks）(参见条款 M23)。事实是大部分人包括编译器生产商在异常处理方面几乎没有什么经验，所以尽管我们知道异常确实会带来开销，却很难预测出开销的准确数量。

谨慎的做法是对本条款所叙述的开销有了解，但是不深究具体的数量（即定性不定量译者注）。不论异常处理的开销有多大我们都得坚持只有必须付出时才付出的原则。为了使你的异常开销最小化，只要可能就尽量采用不支持异常的方法编译程序，把使用 try 块和异常规格限制在你确实需要它们的地方，并且只有在确为异常的情况下（exceptional）才抛出异常。如果你在性能上仍旧有问题，总体评估一下你的软件以决定异常支持是否是一个起作用的因素。如果是，那就考虑选择其它的编译器，能在 C++异常处理方面具有更高实现效率的编译器。

6. 效率

我怀疑一些人在 C++软件开发人员身上进行秘密的巴甫洛夫试验，否则为什么当提到“效率”这个词时，许多程序员都会流口水。（Scott Meyers 真幽默　译者注）

事实上，效率可不是一个开玩笑的事情。一个太大或太慢的程序它们的优点无论多么引人注目都不会为人们所接受。本来就应该这样。软件是用来帮助我们更好地工作，说运行速度慢才是更好的，说需要 32MB 内存的程序比仅仅需要 16MB 内存的程序好，说占用 100MB 磁盘空间的程序比仅仅占用 50MB 磁盘空间的程序好，这简直是无稽之谈。而且尽管有一些程序确是为了进行更复杂的运算才占用更多的时间和空间，但是对于许多程序来说只能归咎于其糟糕的设计和马虎的编程。

在用 C++写出高效地程序之前，必须认识到 C++本身绝对与你所遇到的任何性能上的问题无关。如果想写出一个高效的 C++程序，你必须首先能写出一个高效的算法。太多的开发人员都忽视了这个简单的道理。是的，循环能够被手工展开，移位操作（shift operation）能够替换乘法，但是如果你所使用的高层算法其内在效率很低，这些微调就不会有任何作用。当线性算法可用时你是否还用二次方程式算法？你是否一遍又一遍地计算重复的数值？如果是的话，可以毫不夸张地把你的程序比喻成一个二流的观光地，即如果你有额外的时间，才值得去看一看。

本章的内容从两个角度阐述效率的问题。第一是从语言独立的角度，关注那些你能在任何语言里都能使用的东西。C++为它们提供了特别吸引人的实现途径，因为它对封装的支持非常好，从而能够用更好的算法与数据结构来替代低效的类似实现，同时接口可以保持不变。

第二是关注 C++语言本身。高性能的算法与数据结构虽然非常好，但如果实际编程中代码实现得很粗糙，效率也会降低得相当多。潜在危害性最大的错误是既容易犯而又不容易察觉的错误，濒繁地构造和释放大量的对象就是一种这样的错误。过多的对象构造和对象释放

对于你的程序性能来说就象是在大出血，在每次建立和释放不需要的对象的过程中，宝贵的时间就这么流走了。这个问题在 C++程序中很普遍，我将用四个条款来说明这些对象从哪里来的，在不影响程序代码正确性的基础上又如何消除它们。

建立大量的对象不会使程序变大而只会使其运行速度变慢。还有其它一些影响性能提高的因素，包括程序库的选择和语言特性的实现（implementations of language features）。在下面的条款中我也将涉及。

在学习了本章内容以后，你将熟悉能够提高程序性能的几个原则，这些原则可以适用于你所写的任何程序。你将知道如何准确地防止在你的软件里出现不需要的对象，并且对编译器生成可执行代码的行为有着敏锐的感觉。

俗话说有备无患（forewarned is forearmed）。所以把下面的内容想成是战斗前的准备。

6.1 Item M16：牢记 80－20 准则（80－20 rule）

80－20 准则说的是大约 20％的代码使用了 80％的程序资源；大约 20%的代码耗用了大约 80％的运行时间；大约 20％的代码使用了 80％的内存；大约 20％的代码执行 80％的磁盘访问；80％的维护投入于大约 20％的代码上；通过无数台机器、操作系统和应用程序上的实验这条准则已经被再三地验证过。80－20 准则不只是一条好记的惯用语，它更是一条有关系统性能的指导方针，它有着广泛的适用性和坚实的实验基础。

当想到 80－20 准则时，不要在具体数字上纠缠不清，一些人喜欢更严格的 90－10 准则，而且也有一些试验证据支持它。不管准确地数字是多少，基本的观点是一样的：软件整体的性能取决于代码组成中的一小部分。

当程序员力争最大化提升软件的性能时，80－20 准则既简化了你的工作又使你的工作变得复杂。一方面 80－20 准则表示大多数时间你能够编写性能一般的代码，因为 80％的时间里这些代码的效率不会影响到整个系统的性能，这会减少一些你的工作压力。而另一方面这条准则也表示如果你的软件出现了性能问题，你将面临一个困难的工作，因为你不仅必须找到导致问题的那一小块代码的位置，还必须寻找方法提高它们的性能。这些任务中最困难的一般是找到系统瓶颈。基本上有两个不同的方法用来寻找：大多数人用的方法和正确的方法。

大多数人寻找瓶颈的方法就是猜。通过经验、直觉、算命纸牌、显灵板、传闻或者其它更荒唐的东西，一个又一个程序员一本正经地宣称程序的性能问题已被找到，因为网络的延迟，不正确的内存分配，编译器没有进行足够的优化或者一些笨蛋主管拒绝在关键的循环里使用汇编语句。这些评估总是以一种带有嘲笑的盛气凌人的架式发布出来，通常这些嘲笑者和他们的预言都是错误的。

大多数程序员在他们程序性能特征上的直觉都是错误的，因为程序性能特征往往不能靠

直觉来确定。结果为提高程序各部分的效率而倾注了大量的精力，但是对程序的整体行为没有显著的影响。例如在程序里使用能够最小化计算量的奇特算法和数据结构，但是如果程序的性能限制主要在 I/O 上（I/O-bound）那么就丝毫起不到作用。采用 I/O 性能强劲的程序库代替编译器本身附加的程序库（参见条款 M23），如果程序的性能瓶颈主要在 CPU 上（CPU-bound），这种方法也不会起什么作用。

在这种情况下，面对运行速度缓慢或占用过多内存的程序，你该如何做呢？80－20 准则的含义是：胡乱地提高一部分程序的效率不可能有很大帮助。程序性能特征往往不能靠直觉确定，这个事实意味着试图猜出性能瓶颈不可能比胡乱地提高一部分程序的效率这种方法好到哪里去。那么会后什么结果呢？

结果是用经验猜测程序那 20％的部分只会导致你心痛。正确的方法是用 profiler 程序识别出令人讨厌的程序的 20％部分。不是所有的工作都让 profiler 去做。你想让它去直接地测量你感兴趣的资源。例如如果程序太缓慢，你想让 profiler 告诉你程序的各个部分都耗费了多少时间。然后你关注那些局部效率能够被极大提高的地方，这也将会很大地提高整体的效率。

profiler 告诉你每条语句执行了多少次或各函数被调用了多少次，这是一个作用有限的工具。从提高性能的观点来看，你不用关心一条语句或一个函数被调用了多少次。毕竟很少遇到用户或程序库的调用者抱怨执行了太多的语句或调用了太多的函数。如果软件足够快，没有人关心有多少语句被执行，如果程序运行过慢，不会有人关心语句有多么的少。他们所关心的是他们厌恶等待，如果你的程序让他们等待，他们也会厌恶你。

不过，知道语句执行或函数调用的频繁程度，有时能帮助你洞察软件内部的行为。例如如果你建立了 100 个某种类型的对象，会发现你调用该类的构造函数有上千次，这个信息无疑是有价值的。而且语句和函数的调用次数能间接地帮助你理解不能直接测量的软件行为。例如，如果你不能直接测量动态内存的使用，那么知道内存分配函数和内存释函数的调用频率也是有帮助的。（也就是，operators new, new[], delete, and delete[]—参见条款 M8）

当然即使最好的 profiler 也是受其处理的数据所影响。如果用缺乏代表性的数据 profile 你的程序，你就不能抱怨 profiler 导致你优化程序的那 80％的部分，从而不曾对程序通常的性能有什么影响。记住 profiler 仅能够告诉你在某一次运行（或某几次运行）时一个程序运行情况，所以如果你用不具有代表性的输入数据 profile 一个程序，那你所进行的 profile 也没有代表型。相反这样做很可能导致你去优化不常用的软件行为，而在软件的常用领域，则对软件整体的效率起相反作用（即效率下降）。

防止这种不正确的结果，最好的方法是用尽可能多的数据 profile 你的软件。此外，你必须确保每组数据在客户（或至少是最重要的客户）如何使用软件的方面能有代表性。通常获取有代表性的数据是很容易的，因为许多客户都愿意让你用他们的数据进行 profile。毕竟你是为了他们需求而优化软件。

6.2 Item M17：考虑使用 lazy evaluation（懒惰计算法）

从效率的观点来看，最佳的计算就是根本不计算，那好，不过如果你根本就不用进行计算的话，为什么还在程序开始处加入代码进行计算呢?并且如果你不需要进行计算，那么如何必须执行这些代码呢？

关键是要懒惰。

还记得么？当你还是一个孩子时，你的父母叫你整理房间。你如果象我一样，就会说“好的“，然后继续做你自己的事情。你不会去整理自己的房间。在你心里整理房间被排在了最后的位置，实际上直到你听见父母下到门厅来查看你的房间是否已被整理时，你才会猛跑进自己的房间里并用最快的速度开始整理。如果你走运，你父母可能不会来检查你的房间，那样的话你就能根本不用整理房间了。

同样的延迟策略也适用于具有五年工龄的 C++程序员的工作上。在计算机科学中，我们尊称这样的延迟为 **lazy evaluation**（懒惰计算法）。当你使用了 **lazy evaluation** 后，采用此种方法的类将推迟计算工作直到系统需要这些计算的结果。如果不需要结果，将不用进行计算，软件的客户和你的父母一样，不会那么聪明。

也许你想知道我说的这些到底是什么意思。也许举一个例子可以帮助你理解。**lazy evaluation** 广泛适用于各种应用领域，所以我将分四个部分讲述。

I 引用计数

```
class String { ... };              // 一个 string 类 (the standard
                                   // string type may be implemented
                                   // as described below, but it
                                   // doesn't have to be)
String s1 = "Hello";
String s2 = s1;                    / 调用 string 拷贝构造函数
```

通常 **string** 拷贝构造函数让 **s2** 被 **s1** 初始化后，**s1** 和 **s2** 都有自己的”**Hello**”拷贝。这种拷贝构造函数会引起较大的开销，因为要制作 **s1** 值的拷贝，并把值赋给 **s2**，这通常需要用 **new** 操作符分配堆内存(参见条款 **8**)，需要调用 **strcpy** 函数拷贝 **s1** 内的数据到 **s2**。这是一个 **eager evaluation**（热情计算）：只因为到 **string** 拷贝构造函数，就要制作 **s1** 值的拷贝并把它赋给 **s2**。然而这时的 **s2** 并不需要这个值的拷贝，因为 **s2** 没有被使用。

懒惰能就是少工作。不应该赋给 **s2** 一个 **s1** 的拷贝，而是让 **s2** 与 **s1** 共享一个值。我们只须做一些记录以便知道谁在共享什么，就能够省掉调用 **new** 和拷贝字符的开销。事实上 **s1** 和 **s2** 共享一个数据结构，这对于 **client** 来说是透明的，对于下面的例子来说，这没有什么差别，因为它们只是读数据：

```
cout << s1;                        // 读 s1 的值
cout << s1 + s2;                   // 读 s1 和 s2 的值
```

仅仅当这个或那个 **string** 的值被修改时，共享同一个值的方法才会造成差异。仅仅修改一个 **string** 的值，而不是两个都被修改，这一点是极为重要的。例如这条语句：

```
s2.convertToUpperCase();
```

这是至关紧要的，仅仅修改 **s2** 的值，而不是连 **s1** 的值一块修改。

为了这样执行语句，**string** 的 **convertToUpperCase** 函数应该制作 **s2** 值的一个拷贝，在修改前把这个私有的值赋给 **s2**。在 **convertToUpperCase** 内部，我们不能再懒惰了：必须为 **s2**（共享的）值制作拷贝以让 **s2** 自己使用。另一方面，如果不修改 **s2**，我们就不用制作它自己值的拷贝。继续保持共享值直到程序退出。如果我们很幸运，**s2** 不会被修改，这种情况下我们永远也不会为赋给它独立的值耗费精力。

这种共享值方法的实现细节（包括所有的代码）在条款 **M29** 中被提供，但是其蕴含的原则就是 **lazy evaluation**：除非你确实需要，不去为任何东西制作拷贝。我们应该是懒惰的，只要可能就共享使用其它值。在一些应用领域，你经常可以这么做。

l 区别对待读取和写入

继续讨论上面的 **reference-counting string** 对象。来看看使用 **lazy evaluation** 的第二种方法。考虑这样的代码：

```
String s = "Homer's Iliad";          // 假设是一个
                                     // reference-counted string
...
cout << s[3];                        // 调用 operator[] 读取 s[3]
s[3] = 'x';                          // 调用 operator[] 写入 s[3]
```

首先调用 **operator[]**用来读取 **string** 的部分值，但是第二次调用该函数是为了完成写操作。我们应能够区别对待读调用和写调用，因为读取 **reference-counted string** 是很容易的，而写入这个 **string** 则需要在写入前对该 **string** 值制作一个新拷贝。

我们陷入了困难之中。为了能够这样做，需要在 **operator[]**里采取不同的措施（根据是为了完成读取操作而调用该函数还是为了完成写入操作而调用该函数）。我们如果判断调用 **operator[]**的 **context** 是读取操作还是写入操作呢？残酷的事实是我们不可能判断出来。通过使用 **lazy evaluation** 和条款 **M30** 中讲述的 **proxy class**，我们可以推迟做出是读操作还是写操作的决定，直到我们能判断出正确的答案。

l Lazy Fetching（懒惰提取）

第三个 **lazy evaluation** 的例子，假设你的程序使用了一些包含许多字段的大型对象。这些对象的生存期超越了程序运行期，所以它们必须被存储在数据库里。每一个对都有一个唯一的对象标识符，用来从数据库中重新获得对象：

```
class LargeObject {                  // 大型持久对象
```

```
public:
  LargeObject(ObjectID id);                 // 从磁盘中恢复对象
  const string& field1() const;             // field 1 的值
  int field2() const;                       // field 2 的值
  double field3() const;                    // ...
  const string& field4() const;
  const string& field5() const;
  ...
};
```

现在考虑一下从磁盘中恢复 **LargeObject** 的开销：

```
void restoreAndProcessObject(ObjectID id)
{
  LargeObject object(id);                   // 恢复对象
  ...
}
```

因为 **LargeObject** 对象实例很大，为这样的对象获取所有的数据，数据库的操作的开销将非常大，特别是如果从远程数据库中获取数据和通过网络发送数据时。而在这种情况下，不需要读去所有数据。例如，考虑这样一个程序：

```
void restoreAndProcessObject(ObjectID id)
{
  LargeObject object(id);
  if (object.field2() == 0) {
    cout << "Object " << id << ": null field2.\n";
  }
}
```

这里仅仅需要 **filed2** 的值，所以为获取其它字段而付出的努力都是浪费。

当 **LargeObject** 对象被建立时，不从磁盘上读取所有的数据，这样懒惰法解决了这个问题。不过这时建立的仅是一个对象“壳”，当需要某个数据时，这个数据才被从数据库中取回。这种“**demand-paged**”对象初始化的实现方法是**:**

```
class LargeObject {
public:
  LargeObject(ObjectID id);
  const string& field1() const;
  int field2() const;
```

```
  double field3() const;
  const string& field4() const;
  ...
private:
  ObjectID oid;
  mutable string *field1Value;                    //参见下面有关
  mutable int *field2Value;                       // "mutable"的讨论
  mutable double *field3Value;
  mutable string *field4Value;
  ...
};
LargeObject::LargeObject(ObjectID id)
: oid(id), field1Value(0), field2Value(0), field3Value(0), ...
{}
const string& LargeObject::field1() const
{
  if (field1Value == 0) {
    从数据库中为 filed 1 读取数据，使
    field1Value 指向这个值;
  }
  return *field1Value;
}
```

对象中每个字段都用一个指向数据的指针来表示，LargeObject 构造函数把每个指针初始化为空。这些空指针表示字段还没有从数据库中读取数值。每个 LargeObject 成员函数在访问字段指针所指向的数据之前必须字段指针检查的状态。如果指针为空，在对数据进行操作之前必须从数据库中读取对应的数据。

实现 Lazy Fetching 时，你面临着一个问题：在任何成员函数里都有可能需要初始化空指针使其指向真实的数据，包括在 const 成员函数里，例如 field1。然而当你试图在 const 成员函数里修改数据时，编译器会出现问题。最好的方法是声明字段指针为 mutable，这表示在任何函数里它们都能被修改，甚至在 const 成员函数里（参见 Effective C++条款 21）。这就是为什么在 LargeObject 里把字段声明为 mutable。

关键字 mutalbe 是一个比较新的 C++ 特性，所以你用的编译器可能不支持它。如果是这样，你需要找到另一种方法让编译器允许你在 const 成员函数里修改数据成员。一种方法叫做“fake this”（伪造 this 指针）,你建立一个指向 non-const 指针，指向的对象与 this

指针一样。当你想修改数据成员时，你通过“fake this”访问它：

```
class LargeObject {
public:
  const string& field1() const;              // 没有变化
  ...

private:
  string *field1Value;                       // 不声明为 mutable
  ...                                        // 因为老的编译器不
};                                           // 支持它

const string& LargeObject::field1() const
{
  // 声明指针，fakeThis，其与 this 指向同样的对象
  // 但是已经去掉了对象的常量属性
  LargeObject * const fakeThis =
    const_cast<LargeObject* const>(this);
  if (field1Value == 0) {
    fakeThis->field1Value =                  // 这赋值是正确的，
      the appropriate data                   // 因为 fakeThis 指向的
      from the database;                     //对象不是 const
  }
  return *field1Value;
}
```

这个函数使用了 const_cast(参见条款 2)，去除了*this 的 const 属性。如果你的编译器不支持 cosnt_cast，你可以使用老式 C 风格的 cast：

```
// 使用老式的 cast，来模仿 mutable
const string& LargeObject::field1() const
{
  LargeObject * const fakeThis = (LargeObject* const)this;
  ...                                        // as above
}
```

再来看 LargeObject 里的指针，必须把这些指针都初始化为空，然后每次使用它们时都必须进行测试，这是令人厌烦的而且容易导致错误发生。幸运的是使用 smart(灵巧)指针可

以自动地完成这种苦差使，具体内容可以参见条款 M28。如果在 LargeObject 里使用 smart 指针，你也将发现不再需要用 mutalbe 声明指针。这只是暂时的，因为当你实现 smart 指针类时你最终会碰到 mutalbe。

I Lazy Expression Evaluation(懒惰表达式计算)

有关 lazy evaluation 的最后一个例子来自于数字程序。考虑这样的代码：

```
template<class T>
class Matrix { ... };                          // for homogeneous matrices
Matrix<int> m1(1000, 1000);                    // 一个 1000 * 1000 的矩阵
Matrix<int> m2(1000, 1000);                    // 同上
...
Matrix<int> m3 = m1 + m2;                      // m1＋m2
```

通常 operator 的实现使用 eagar evaluation：在这种情况下，它会计算和返回 m1 与 m2 的和。这个计算量相当大（1000000 次加法运算），当然系统也会分配内存来存储这些值。

lazy evaluation 方法说这样做工作太多，所以还是不要去做。而是应该建立一个数据结构来表示 m3 的值是 m1 与 m2 的和，在用一个 enum 表示它们间是加法操作。很明显，建立这个数据结构比 m1 与 m2 相加要快许多，也能够节省大量的内存。

考虑程序后面这部分内容，在使用 m3 之前，代码执行如下：

```
Matrix<int> m4(1000, 1000);
...                                            // 赋给 m4 一些值
m3 = m4 * m1;
```

现在我们可以忘掉 m3 是 m1 与 m2 的和（因此节省了计算的开销），在这里我们应该记住 m3 是 m4 与 m1 运算的结果。不必说，我们不用进行乘法运算。因为我们是懒惰的，还记得么？

这个例子看上去有些做作，因为一个好的程序员不会这样写程序：计算两个矩阵的和而不去用它们，但是它实际上又不象看上去的那么做作。虽然好程序员不会进行不需要的计算，但是在维护中程序员修改了程序的路径，使得以前有用的计算变得没有了作用，这种情况是常见的。通过定义使用前才进行计算的对象可以减少这种情况发生的可能性（参见 Effective C++条款 32），不过这个问题偶尔仍然会出现。

但是如果这就是使用 lazy evaluation 唯一的时机，那就太不值得了。一个更常见的应用领域是当我们仅仅需要计算结果的一部分时。例如假设我们初始化 m3 的值为 m1 和 m2 的和，然后象这样使用 m3：

```
cout << m3[4];                                 // 打印 m3 的第四行
```

很明显，我们不能再懒惰了，应该计算 m3 的第四行值。但是我们也不能雄心过大，我们没有理由计算 m3 第四行以外的结果；m3 其余的部分仍旧保持未计算的状态直到确实需要

它们的值。很走运，我们一直不需要。

我们怎么可能这么走运呢？矩阵计算领域的经验显示这种可能性很大。实际上 **lazy evaluation** 就存在于 **APL** 语言中。**APL** 是在 **1960** 年代发展起来语言，能够进行基于矩阵的交互式的运算。那时侯运行它的计算机的运算能力还没有现在微波炉里的芯片高，**APL** 表面上能够进行进行矩阵的加、乘，甚至能够快速地与大矩阵相除！它的技巧就是 **lazy evaluation**。这个技巧通常是有效的，因为一般 **APL** 的用户加、乘或除以矩阵不是因为他们需要整个矩阵的值，而是仅仅需要其一小部分的值。**APL** 使用 **lazy evaluation** 来拖延它们的计算直到确切地知道需要矩阵哪一部分的结果，然后仅仅计算这一部分。实际上，这能允许用户在一台根本不能完成 **eager evaluation** 的计算机上交互式地完成大量的计算。现在计算机速度很快，但是数据集也更大，用户也更缺乏耐心，所以很多现在的矩阵库程序仍旧使用 **lazy evaluation**。

公正地讲，懒惰有时也会失败。如果这样使用 **m3**：

```
cout << m3;                                    // 打印 m3 所有的值
```

一切都完了，我们必须计算 **m3** 的全部数值。同样如果修改 **m3** 所依赖的任一个矩阵，我们也必须立即计算：

```
m3 = m1 + m2;                                  // 记住 m3 是 m1 与 m2 的和
                                               //
m1 = m4;                                       // 现在 m3 是 m2 与 m1 的旧值之和！
                                               //
```

这里我们我们必须采取措施确保赋值给 **m1** 以后不会改变 **m3**。在 **Matrix<int>**赋值操作符里，我们能够在改变 **m1** 之前捕获 **m3** 的值，或者我们可以给 **m1** 的旧值制作一个拷贝让 **m3** 依赖于这个拷贝计算，我们必须采取措施确保 **m1** 被赋值以后 **m3** 的值保持不变。其它可能会修改矩阵的函数都必须用同样的方式处理。

因为需要存储两个值之间的依赖关系，维护存储值、依赖关系或上述两者，重载操作符例如赋值符、拷贝操作和加法操作，所以 **lazy evaluation** 在数字领域应用得很多。另一方面运行程序时它经常节省大量的时间和空间。

I 总结

以上这四个例子展示了 **lazy evaluation** 在各个领域都是有用的：能避免不需要的对象拷贝，通过使用 **operator[]**区分出读操作，避免不需要的数据库读取操作，避免不需要的数字操作。但是它并不总是有用。就好象如果你的父母总是来检查你的房间，那么拖延整理房间将不会减少你的工作量。实际上，如果你的计算都是重要的，**lazy evaluation** 可能会减慢速度并增加内存的使用，因为除了进行所有的计算以外，你还必须维护数据结构让 **lazy evaluation** 尽可能地在第一时间运行。在某些情况下要求软件进行原来可以避免的计算，这时 **lazy evaluation** 才是有用的。

lazy evaluation 对于 **C++**来说没有什么特殊的东西。这个技术能被运用于各种语言里，几种语言例如著名的 **APL**、**dialects of Lisp**（事实上所有的数据流语言）都把这种思想做为语言的一个基本部分。然而主流程序设计语言采用的是 **eager evaluation**，**C++**是主流语言。不过 **C++**特别适合用户实现 **lazy evaluation**，因为它对封装的支持使得能在类里加入 **lazy evaluation**，而根本不用让类的使用者知道。

再看一下上述例子中的代码片段，你就能知道采用 **eager** 还是 **lazy evaluation**，在类提供的接口中并没有半点差别。这就是说我们可以直接用 **eager evaluation** 方法来实现一个类，但是如果你用通过 **profiler** 调查（参见条款 **M16**）显示出类实现有一个性能瓶颈，就可以用使用 **lazy evaluation** 的类实现来替代它（参见 **Effective C++**条款 **34**）。对于使用者来说所改变的仅是性能的提高（重新编译和链接后）。这是使用者喜欢的软件升级方式，它使你完全可以为懒惰而骄傲。

6.3 Item M18：分期摊还期望的计算

在条款 **M17** 中，我极力称赞懒惰的优点，尽可能地拖延时间，并且我解释说懒惰如何提高程序的运行效率。在这个条款里我将采用一种不同的态度。这里将不存在懒惰。我鼓励你让程序做的事情比被要求的还要多，通过这种方式来提高软件的性能。这个条款的核心就是 **over-eager evaluation**（过度热情计算法）：在要求你做某些事情以前就完成它们。例如下面这个模板类，用来表示放有大量数字型数据的一个集合：

```
template<class NumericalType>
class DataCollection {
public:
  NumericalType min() const;
  NumericalType max() const;
  NumericalType avg() const;
  ...
};
```

假设 **min,max** 和 **avg** 函数分别返回现在这个集合的最小值，最大值和平均值，有三种方法实现这三种函数。使用 **eager evaluation(**热情计算法**)**，当 **min,max** 和 **avg** 函数被调用时，我们检测集合内所有的数值，然后返回一个合适的值。使用 **lazy evaluation**（懒惰计算法）**,**只有确实需要函数的返回值时我们才要求函数返回能用来确定准确数值的数据结构。使用 **over-eager evaluation**（过度热情计算法），我们随时跟踪目前集合的最小值，最大值和平均值，这样当 **min,max** 或 **avg** 被调用时，我们可以不用计算就立刻返回正确的数值。如果频繁调用 **min,max** 和 **avg**，我们把跟踪集合最小值、最大值和平均值的开销分摊到所有这些函数的调用上，每次函数调用所分摊的开销比 **eager evaluation** 或 **lazy evaluation**

要小。

隐藏在 over-eager evaluation 后面的思想是如果你认为一个计算需要频繁进行，你就可以设计一个数据结构高效地处理这些计算需求，这样可以降低每次计算需求时的开销。

采用 over-eager 最简单的方法就是 caching(缓存)那些已经被计算出来而以后还有可能需要的值。例如你编写了一个程序，用来提供有关雇员的信息，这些信息中的经常被需要的部分是雇员的办公隔间号码。而假设雇员信息存储在数据库里，但是对于大多数应用程序来说，雇员隔间号都是不相关的，所以数据库不对查抄它们进行优化。为了避免你的程序给数据库造成沉重的负担，可以编写一个函数 findCubicleNumber，用来缓存查找到的数据。以后需要已经被获取的隔间号时，可以在 cache 里找到，而不用向数据库查询。

以下是实现 findCubicleNumber 的一种方法：它使用了标准模板库（STL）里的 map 对象（有关 STL 参见条款 M35）。

```
int findCubicleNumber(const string& employeeName)
{
  // 定义静态 map，存储 (employee name, cubicle number)
  // pairs. 这个 map 是 local cache。
  typedef map<string, int> CubicleMap;
  static CubicleMap cubes;
  // try to find an entry for employeeName in the cache;
  // the STL iterator "it" will then point to the found
  // entry, if there is one (see Item 35 for details)
  CubicleMap::iterator it = cubes.find(employeeName);
  // "it"'s value will be cubes.end() if no entry was
  // found (this is standard STL behavior). If this is
  // the case, consult the database for the cubicle
  // number, then add it to the cache
  if (it == cubes.end()) {
    int cubicle =
      the result of looking up employeeName's cubicle
      number in the database;
    cubes[employeeName] = cubicle;          // add the pair
                                            // (employeeName, cubicle)
                                            // to the cache
    return cubicle;
  }
```

```
  else {
    // "it" points to the correct cache entry, which is a
    // (employee name, cubicle number) pair. We want only
    // the second component of this pair, and the member
    // "second" will give it to us
    return (*it).second;
  }
}
```

不要陷入 **STL** 代码的实现细节里（你读完条款 **M35** 以后，你会比较清楚）。应该把注意力放在这个函数蕴含的方法上。这个方法是使用 **local cache**，用开销相对不大的内存中查询来替代开销较大的数据库查询。假如隔间号被不止一次地频繁需要，在 **findCubicleNumber** 内使用缓存会减少返回隔间号的平均开销。

（上述代码里有一个细节需要解释一下，最后一个语句返回的是**(*it).second**，而不是常用的 **it->second**。为什么？答案是这是为了遵守 **STL** 的规则。简单地说，**iterator** 是一个对象，不是指针，所以不能保证”**->**”被正确应用到它上面。不过 **STL** 要求”**.**”和”*****”在 **iterator** 上是合法的，所以**(*it).second** 在语法上虽然比较繁琐，但是保证能运行。）

catching 是一种分摊期望的计算开销的方法。**Prefetching(**预提取**)**是另一种方法。你可以把 **prefech** 想象成购买大批商品而获得的折扣。例如磁盘控制器从磁盘读取数据时，它们会读取一整块或整个扇区的数据，即使程序仅需要一小块数据。这是因为一次读取一大块数据比在不同时间读取两个或三个小块数据要快。而且经验显示如果需要一个地方的数据，则很可能也需要它旁边的数据。这是位置相关现象。正因为这种现象，系统设计者才有理由为指令和数据使用磁盘 **cache** 和内存 **cache**，还有使用指令 **prefetch**。

你说你不关心象磁盘控制器或 **CPU cache** 这样低级的东西。没有问题，**prefetch** 在高端应用里也有优点。例如你为 **dynamic** 数组实现一个模板，**dynamic** 就是开始时具有一定的尺寸，以后可以自动扩展的数组，所以所有非负的索引都是合法的：

```
template<class T>                       // dynamic 数组
class DynArray { ... };                 // 模板

DynArray<double> a;                     // 在这时，只有 a[0]
                                        // 是合法的数组元素
a[22] = 3.5;                            // a 自动扩展
                                        //: 现在索引 0－22
                                        // 是合法的
a[32] = 0;                            // 有自行扩展;
```

```
                                        // 现在 a[0]-a[32]是合法的
```

一个 DynArray 对象如何在需要时自行扩展呢？一种直接的方法是分配所需的额外的内存。就象这样：

```
template<class T>
T& DynArray<T>::operator[](int index)
{
  if (index < 0) {
    throw an exception;                    // 负数索引仍不合法
  }
  if (index >当前最大的索引值) {
    调用 new 分配足够的额外内存，以使得
   索引合法;
  }
  返回 index 位置上的数组元素;
}
```

每次需要增加数组长度时，这种方法都要调用 new，但是调用 new 会触发 operator new(参见条款 M8)，operator new (和 operator delete)的调用通常开销很大。因为它们将导致底层操作系统的调用，系统调用的速度一般比进程内函数调用的速度慢。因此我们应该尽量少使用系统调用。

使用 Over-eager evaluation 方法，其原因我们现在必须增加数组的尺寸以容纳索引 i，那么根据位置相关性原则我们可能还会增加数组尺寸以在未来容纳比 i 大的其它索引。为了避免为扩展而进行第二次（预料中的）内存分配，我们现在增加 DynArray 的尺寸比能使 i 合法的尺寸要大，我们希望未来的扩展将被包含在我们提供的范围内。例如我们可以这样编写 DynArray::operator[]：

```
template<class T>
T& DynArray<T>::operator[](int index)
{
  if (index < 0) throw an exception;
  if (index > 当前最大的索引值) {
    int diff = index – 当前最大的索引值;
    调用 new 分配足够的额外内存，使得 index+diff 合法;
  }
  返回 index 位置上的数组元素;
}
```

这个函数每次分配的内存是数组扩展所需内存的两倍。如果我们再来看一下前面遇到的那种情况，就会注意到 DynArray 分配了一次额外内存，即使它的逻辑尺寸被扩展了两次：

```
DynArray<double> a;                    // 仅仅 a[0]是合法的
a[22] = 3.5;                           // 调用 new 扩展
                                       // a 的存储空间到索引 44
                                       // a 的逻辑尺寸
                                       // 变为 23
a[32] = 0;                             // a 的逻辑尺寸
                                       // 被改变，允许使用 a[32],
                                       // 但是没有调用 new
```

如果再次需要扩展 a，只要提供的新索引不大于 44，扩展的开销就不大。

贯穿本条款的是一个常见的主题，更快的速度经常会消耗更多的内存。跟踪运行时的最小值、最大值和平均值，这需要额外的空间，但是能节省时间。Cache 运算结果需要更多的内存，但是一旦需要被缓存的结果时就能减少需要重新生成的时间。Prefetch 需要空间放置被 prefetch 的东西，但是它减少了访问它们所需的时间。自从有了计算机就有这样的描述：你能以空间换时间。（然而不总是这样，使用大型对象意味着不适合虚拟内存或 cache 页。在一些罕见的情况下，建立大对象会降低软件的性能，因为分页操作的增加（详见操作系统中内存管理 译者注），cache 命中率降低，或者两者都同时发生。如何发现你正遭遇这样的问题呢？你必须 profile, profile, profile(参见条款 M16)。

在本条款中我提出的建议，即通过 over-eager 方法分摊预期计算的开销，例如 caching 和 prefething，这并不与我在条款 M17 中提出的有关 lazy evaluation 的建议相矛盾。当你必须支持某些操作而不总需要其结果时，lazy evaluation 是在这种时候使用的用以提高程序效率的技术。当你必须支持某些操作而其结果几乎总是被需要或被不止一次地需要时，over-eager 是在这种时候使用的用以提高程序效率的一种技术。它们所产生的巨大的性能提高证明在这方面花些精力是值得的。

6.4 Item M19：理解临时对象的来源

当程序员之间进行交谈时，他们经常把仅仅需要一小段时间的变量称为临时变量。例如在下面这段 swap(交换)例程里：

```
template<class T>
void swap(T& object1, T& object2)
{
  T temp = object1;
  object1 = object2;
```

```
  object2 = temp;
}
```

通常把 **temp** 叫做临时变量。不过就 **C++**而言，**temp** 根本不是临时变量，它只是一个函数的局部对象。

在 **C++**中真正的临时对象是看不见的，它们不出现在你的源代码中。建立一个没有命名的非堆（**non-heap**）对象会产生临时对象。这种未命名的对象通常在两种条件下产生：为了使函数成功调用而进行隐式类型转换和函数返回对象时。理解如何和为什么建立这些临时对象是很重要的，因为构造和释放它们的开销对于程序的性能来说有着不可忽视的影响。

首先考虑为使函数成功调用而建立临时对象这种情况。当传送给函数的对象类型与参数类型不匹配时会产生这种情况。例如一个函数，它用来计算一个字符在字符串中出现的次数：

```
// 返回 ch 在 str 中出现的次数
size_t countChar(const string& str, char ch);
char buffer[MAX_STRING_LEN];
char c;
// 读入到一个字符和字符串中，用 setw
// 避免缓存溢出，当读取一个字符串时
cin >> c >> setw(MAX_STRING_LEN) >> buffer;
cout << "There are " << countChar(buffer, c)
     << " occurrences of the character " << c
     << " in " << buffer << endl;
```

看一下 **countChar** 的调用。第一个被传送的参数是字符数组，但是对应函数的正被绑定的参数的类型是 **const string&**。仅当消除类型不匹配后，才能成功进行这个调用，你的编译器很乐意替你消除它，方法是建立一个 **string** 类型的临时对象。通过以 **buffer** 做为参数调用 **string** 的构造函数来初始化这个临时对象。**countChar** 的参数 **str** 被绑定在这个临时的 **string** 对象上。当 **countChar** 返回时，临时对象自动释放。

这样的类型转换很方便（尽管很危险－参见条款 **M5**），但是从效率的观点来看，临时 **string** 对象的构造和释放是不必要的开销。通常有两个方法可以消除它。一种是重新设计你的代码，不让发生这种类型转换。这种方法在条款 **M5** 中被研究和分析。另一种方法是通过修改软件而不再需要类型转换，条款 **M21** 讲述了如何去做。

仅当通过传值（**by value**）方式传递对象或传递常量引用（**reference-to-const**）参数时，才会发生这些类型转换。当传递一个非常量引用（**reference-to-non-const**）参数对象，就不会发生。考虑一下这个函数：

```
void uppercasify(string& str);          // 把 str 中所有的字符
                                        // 改变成大写
```

在字符计数的例子里，能够成功传递 char 数组到 countChar 中，但是在这里试图用 char 数组调用 upeercasify 函数，则不会成功：

```
char subtleBookPlug[] = "Effective C++";
uppercasify(subtleBookPlug);                    // 错误!
```

没有为使调用成功而建立临时对象，为什么呢？

假设建立一个临时对象，那么临时对象将被传递到 upeercasify 中，其会修改这个临时对象，把它的字符改成大写。但是对 subtleBookPlug 函数调用的真正参数没有任何影响；仅仅改变了临时从 subtleBookPlug 生成的 string 对象。无疑这不是程序员所希望的。程序员传递 subtleBookPlug 参数到 uppercasify 函数中，期望修改 subtleBookPlug 的值。当程序员期望修改非临时对象时，对非常量引用（references-to-non-const）进行的隐式类型转换却修改临时对象。这就是为什么 C++语言禁止为非常量引用（reference-to-non-const）产生临时对象。这样非常量引用（reference-to-non-const）参数就不会遇到这种问题。

建立临时对象的第二种环境是函数返回对象时。例如 operator+必须返回一个对象，以表示它的两个操作数的和（参见 Effective C++ 条款 23）。例如给定一个类型 Number，这种类型的 operator+被这样声明：

```
const Number operator+(const Number& lhs,
                       const Number& rhs);
```

这个函数的返回值是临时的，因为它没有被命名；它只是函数的返回值。你必须为每次调用 operator+构造和释放这个对象而付出代价。（有关为什么返回值是 const 的详细解释，参见 Effective C++条款 21）

通常你不想付出这样的开销。对于这种函数，你可以切换到 operator=，而避免开销。条款 M22 告诉我们进行这种转换的方法。不过对于大多数返回对象的函数来说，无法切换到不同的函数，从而没有办法避免构造和释放返回值。至少在概念上没有办法避免它。然而概念和现实之间又一个黑暗地带，叫做优化，有时你能以某种方法编写返回对象的函数，以允许你的编译器优化临时对象。这些优化中，最常见和最有效的是返回值优化，这是条款 M20 的内容。

综上所述，临时对象是有开销的，所以你应该尽可能地去除它们，然而更重要的是训练自己寻找可能建立临时对象的地方。在任何时候只要见到常量引用（reference-to-const）参数，就存在建立临时对象而绑定在参数上的可能性。在任何时候只要见到函数返回对象，就会有一个临时对象被建立（以后被释放）。学会寻找这些对象构造，你就能显著地增强透过编译器表面动作而看到其背后开销的能力。

6.5 Item M20：协助完成返回值优化

一个返回对象的函数很难有较高的效率，因为传值返回会导致调用对象内的构造和析构

函数(参见条款 M19)，这种调用是不能避免的。问题很简单：一个函数要么为了保证正确的行为而返回对象要么就不这么做。如果它返回了对象，就没有办法摆脱被返回的对象。就说到这。

考虑 rational(有理数)类的成员函数 operator*：

```
class Rational {
public:
  Rational(int numerator = 0, int denominator = 1);
  ...
  int numerator() const;
  int denominator() const;
};

// 有关为什么返回值是 const 的解释，参见条款 M6,
const Rational operator*(const Rational& lhs,
                         const Rational& rhs);
```

甚至不用看 operator*的代码，我们就知道它肯定要返回一个对象，因为它返回的是两个任意数字的计算结果。这些结果是任意的数字。operator*如何能避免建立新对象来容纳它们的计算结果呢？这是不可能的，所以它必须得建立新对象并返回它。不过 C++程序员仍然花费大量的精力寻找传说中的方法，能够去除传值返回的对象（参见 Effective C++ 条款 23 和条款 31）。

有时人们会返回指针，从而导致这种滑稽的句法：

```
// 一种不合理的避免返回对象的方法
const Rational * operator*(const Rational& lhs,
                           const Rational& rhs);
Rational a = 10;
Rational b(1, 2);
Rational c = *(a * b);              //你觉得这样很“正常”么？
```

它也引发出一个问题。调用者应该删除函数返回对象的指针么？答案通常是肯定的，并且通常会导致资源泄漏。

其它一些开发人员会返回引用。这种方法能产生可接受的句法，

```
//一种危险的(和不正确的)方法，用来避免返回对象
const Rational& operator*(const Rational& lhs,
                          const Rational& rhs);
Rational a = 10;
```

```
Rational b(1, 2);
Rational c = a * b;                              // 看上去很合理
```

但是函数不能被正确地实现。一种尝试的方法是这样的：

```
// 另一种危险的方法 (和不正确的)方法，用来
// 避免返回对象
const Rational& operator*(const Rational& lhs,
                          const Rational& rhs)
{
  Rational result(lhs.numerator() * rhs.numerator(),
                  lhs.denominator() * rhs.denominator());
  return result;//WQ 加注 返回时，其指向的对象已经不存在了
}
```

这个函数返回的引用，其指向的对象已经不存在了。它返回的是一个指向局部对象 **result** 的引用，当 **operator*** 退出时 **result** 被自动释放。返回指向已被释放的对象的引用，这样的引用绝对不能使用。

相信我：一些函数（**operator***也在其中）必须要返回对象。这就是它们的运行方法。不要与其对抗，你不会赢的。

你消除传值返回的对象的努力不会获得胜利。这是一场错误的战争。从效率的观点来看，你不应该关心函数返回的对象，你仅仅应该关心对象的开销。你所应该关心的是把你的努力引导到寻找减少返回对象的开销上来，而不是去消除对象本身（我们现在认识到这种寻求是无用的）。如果没有与这些对象相关的开销，谁还会关心有多少对象被建立呢？

以某种方法返回对象，能让编译器消除临时对象的开销，这样编写函数通常是很普遍的。这种技巧是返回 **constructor argument** 而不是直接返回对象，你可以这样做：

```
// 一种高效和正确的方法，用来实现
// 返回对象的函数
const Rational operator*(const Rational& lhs,
                         const Rational& rhs)
{
  return Rational(lhs.numerator() * rhs.numerator(),
                  lhs.denominator() * rhs.denominator());
}
```

仔细观察被返回的表达式。它看上去好象正在调用 **Rational** 的构造函数，实际上确是这样。你通过这个表达式建立一个临时的 **Rational** 对象，

```
Rational(lhs.numerator() * rhs.numerator(),
```

```
        lhs.denominator() * rhs.denominator());
```

并且这是一个临时对象，函数把它拷贝给函数的返回值。

返回 **constructor argument** 而不出现局部对象，这种方法还会给你带来很多开销，因为你仍旧必须为在函数内临时对象的构造和释放而付出代价，你仍旧必须为函数返回对象的构造和释放而付出代价。但是你已经获得了好处。**C++**规则允许编译器优化不出现的临时对象（**temporary objects out of existence**）。因此如果你在如下的环境里调用 **operator***：

```
Rational a = 10;
Rational b(1, 2);
Rational c = a * b;                          // 在这里调用 operator*
```

编译器就会被允许消除在 **operator*** 内的临时变量和 **operator*** 返回的临时变量。它们能在为目标 **c** 分配的内存里构造 **return** 表达式定义的对象。如果你的编译器这样去做，调用 **operator*** 的临时对象的开销就是零：没有建立临时对象。你的代价就是调用一个构造函数－－建立 **c** 时调用的构造函数。而且你不能比这做得更好了，因为 **c** 是命名对象，命名对象不能被消除（参见条款 **M22**）。不过你还可以通过把函数声明为 **inline** 来消除 **operator*** 的调用开销（不过首先参见 **Effective C++** 条款 **33**）：

```
// the most efficient way to write a function returning
// an object
inline const Rational operator*(const Rational& lhs,
                                const Rational& rhs)
{
  return Rational(lhs.numerator() * rhs.numerator(),
                  lhs.denominator() * rhs.denominator());
}
```

“好，不错”，你嘀咕地说，“优化，谁关心编译器能做什么？我想知道它们确实做了什么，**Does any of this nonsense work with real compilers?**” **It does**。这种特殊的优化－－通过使用函数的 **return** 位置（或者在函数被调用位置用一个对象来替代）来消除局部临时对象－－是众所周知的和被普遍实现的。它甚至还有一个名字：返回值优化（**return value optimization**）（**WQ** 加注：在《深度探索 **C**＋＋物件模型》中有更多更详细的讲述，它叫之为 **named return value optimization**。但注意，这种优化对普通的赋值运算无效，编译器不能够用拷贝构造函数取代赋值运算动作，最终结论是：在确保语意正确的前题下没有更好的优化可能了）。实际上这种优化有自己的名字本身就可以解释为什么它被广泛地使用。寻找 **C++**编译器的程序员会问销售商编译器是否有返回值优化功能。如果一个销售商说有而另一个问“那是什么东西？”，第一个销售商就会有明显的竞争优势。啊，资本主义，有时你实在应该去爱它。（谨代表作者观点，译者坚决拥护四项基本原则　　译者注　**:-**）**)**

附：

文中最后一段黑体部分如何翻译，我有些拿不准，请高手告知，为了容易理解，我在此附上此文最后一段的英文原文：

"Yeah, yeah," you mutter, "optimization, schmoptimization. Who cares what compilers can do? I want to know what they do do. Does any of this nonsense work with real compilers?" It does. This particular optimization — eliminating a local temporary by using a function's return location (and possibly replacing that with an object at the function's call site) — is both well-known and commonly implemented. It even has a name: the return value optimization. In fact, the existence of a name for this optimization may explain why it's so
widely available. Programmers looking for a C++ compiler can ask vendors whether the return value optimization is implemented. If one vendor says yes and another says "The what?," the first vendor has a notable competitive advantage. Ah, capitalism. Sometimes you just gotta love it.

6.6 Item M21：通过重载避免隐式类型转换

以下是一段代码，如果没有什么不寻常的原因，实在看不出什么东西：

```
class UPInt {                                   // unlimited precision
public:                                         // integers 类
  UPInt();
  UPInt(int value);
  ...
};
//有关为什么返回值是 const 的解释，参见 Effective C++ 条款 21
const UPInt operator+(const UPInt& lhs, const UPInt& rhs);
UPInt upi1, upi2;
...
UPInt upi3 = upi1 + upi2;
```

这里还看不出什么令人惊讶的东西。**upi1** 和 **upi2** 都是 **UPInt** 对象，所以它们之间相加就会调用 **UPInts** 的 **operator** 函数。

现在考虑下面这些语句：

```
upi3 = upi1 + 10;
upi3 = 10 + upi2;
```

这些语句也能够成功运行。方法是通过建立临时对象把整形数 **10** 转换为 **UPInts**（参见

条款 M19）。

让编译器完成这种类型转换是确实是很方便，但是建立临时对象进行类型转换工作是有开销的，而我们不想承担这种开销。就象大多数人只想从政府那里受益而不想为此付出一样，大多数 C++程序员希望进行没有临时对象开销的隐式类型转换。但是在计算领域里发生不了赤字现象，我们如何能这么做呢？

让我们回退一步，认识到我们的目的不是真的要进行类型转换，而是用 UPint 和 int 做为参数调用 operator+。隐式类型转换只是用来达到目的的手段，但是我们不要混淆手段与目的。还有一种方法可以成功进行 operator+的混合类型调用，它将消除隐式类型转换的需要。如果我们想要把 UPInt 和 int 对象相加，通过声明如下几个函数达到这个目的，每一个函数有不同的参数类型集。

```
const UPInt operator+(const UPInt& lhs,      // add UPInt
                      const UPInt& rhs);     // and UPInt
const UPInt operator+(const UPInt& lhs,      // add UPInt
                      int rhs);              // and int
const UPInt operator+(int lhs,               // add int and
                      const UPInt& rhs);     // UPInt
UPInt upi1, upi2;
...
UPInt upi3 = upi1 + upi2;                    // 正确,没有由 upi1 或 upi2
                                             // 生成的临时对象
upi3 = upi1 + 10;                            // 正确，没有由 upi1 or 10
                                             // 生成的临时对象
upi3 = 10 + upi2;                            //正确，没有由 10 or upi2
                                             //生成的临时对象。
```

一旦你开始用函数重载来消除类型转换，你就有可能这样声明函数，把自己陷入危险之中：

```
const UPInt operator+(int lhs, int rhs);          // 错误!
```

这个想法是合情合理的。对于 UPInt 和 int 类型，我们想要用所有可能的组合来重载 operator 函数。上面只给出了三种重载函数，唯一漏掉的是带有两个 int 参数的 operator，所以我们想把它加上。

有道理么？在 C++中有一条规则是每一个重载的 operator 必须带有一个用户定义类型（user-defined type）的参数。int 不是用户定义类型，所以我们不能重载 operator 成为仅带有此[int]类型参数的函数。（如果没有这条规则，程序员将能改变预定义的操作，这样做肯定把程序引入混乱的境地。比如企图重载上述的 operator，将会改变 int 类型相加的

含义。）

利用重载避免临时对象的方法不只是用在 operator 函数上。比如在大多数程序中，你想允许在所有能使用 string 对象的地方，也一样可以使用 char*，反之亦然。同样如果你正在使用 numerical（数字）类，例如 complex（参见条款 M35）,你想让 int 和 double 这样的类型可以使用在 numerical 对象的任何地方。因此任何带有 string、char*、complex 参数的函数可以采用重载方式来消除类型转换。

不过，必须谨记 80－20 规则（参见条款 M16）。没有必要实现大量的重载函数，除非你有理由确信程序使用重载函数以后其整体效率会有显著的提高。

6.7 Item M22：考虑用运算符的赋值形式（op=）取代其单独形式（op）

大多数程序员认为如果他们能这样写代码：

```
x = x + y;                    x = x - y;
```

那他们也能这样写：

```
x += y;                       x -= y;
```

如果 x 和 y 是用户定义的类型（user-defined type），就不能确保这样。就 C++来说，operator+、operator=和 operator+=之间没有任何关系，因此如果你想让这三个 operator 同时存在并具有你所期望的关系，就必须自己实现它们。同理，operator -, *, /, 等等也一样。

确保 operator 的赋值形式（assignment version）（例如 operator+=）与一个 operator 的单独形式（stand-alone）(例如 operator+)之间存在正常的关系，一种好方法是后者（指 operator+ 译者注）根据前者（指 operator+= 译者注）来实现（参见条款 M6）。这很容易：

```
class Rational {
public:
  ...
  Rational& operator+=(const Rational& rhs);
  Rational& operator-=(const Rational& rhs);
};

// operator+ 根据 operator+=实现;
//有关为什么返回值是 const 的解释，
//参见 Effective C++条款 21 和 109 页 的有关实现的警告
const Rational operator+(const Rational& lhs,
                         const Rational& rhs)
```

```
{
  return Rational(lhs) += rhs;
}
// operator- 根据 operator -= 来实现
const Rational operator-(const Rational& lhs,
                         const Rational& rhs)
{
  return Rational(lhs) -= rhs;
}
```

在这个例子里，从零开始实现 **operator+=**和**-=**，而 **operator+** 和 **operator-** 则是通过调用前述的函数来提供自己的功能。使用这种设计方法，只用维护 **operator** 的赋值形式就行了。而且如果假设 **operator** 赋值形式在类的 **public** 接口里，这就不用让 **operator** 的单独形式成为类的友元（参见 **Effective C++** 条款 **19**）。

如果你不介意把所有的 **operator** 的单独形式放在全局域里，那就可以使用模板来替代单独形式的函数的编写：

```
template<class T>
const T operator+(const T& lhs, const T& rhs)
{
  return T(lhs) += rhs;                    // 参见下面的讨论
}
template<class T>
const T operator-(const T& lhs, const T& rhs)
{
  return T(lhs) -= rhs;                     // 参见下面的讨论
}
...
```

使用这些模板，只要为 **operator** 赋值形式定义某种类型，一旦需要，其对应的 **operator** 单独形式就会被自动生成。

这样编写确实不错，但是到目前为止，我们还没有考虑效率问题，效率毕竟是本章的主题。在这里值得指出的是三个效率方面的问题。第一、总的来说 **operator** 的赋值形式比其单独形式效率更高，因为单独形式要返回一个新对象，从而在临时对象的构造和释放上有一些开销（参见条款 **M19** 和条款 **M20**，还有 **Effective C++**条款 **23**）。**operator** 的赋值形式把结果写到左边的参数里，因此不需要生成临时对象来容纳 **operator** 的返回值。

第二、提供 **operator** 的赋值形式的同时也要提供其标准形式，允许类的客户端在便利

与效率上做出折衷选择。也就是说，客户端可以决定是这样编写：

```
Rational a, b, c, d, result;
...
result = a + b + c + d;              // 可能用了 3 个临时对象
                                     // 每个 operator+ 调用使用 1 个
```

还是这样编写：

```
result = a;                                  //不用临时对象
result += b;                                 //不用临时对象
result += c;                                 //不用临时对象
result += d;                                 //不用临时对象
```

前者比较容易编写、debug 和维护，并且在 80%的时间里它的性能是可以被接受的（参见条款 M16）。后者具有更高的效率，估计这对于汇编语言程序员来说会更直观一些。通过提供两种方案，你可以让客户端开发人员用更容易阅读的单独形式的 operator 来开发和 debug 代码，同时保留用效率更高的 operator 赋值形式替代单独形式的权力。而且根据 operator 的赋值形式实现其单独形式，这样你能确保当客户端从一种形式切换到另一种形式时，操作的语义可以保持不变。

最后一点，涉及到 operator 单独形式的实现。再看看 operator+ 的实现：

```
template<class T>
const T operator+(const T& lhs, const T& rhs)
{ return T(lhs) += rhs; }
```

表达式 T(lhs)调用了 T 的拷贝构造函数。它建立一个临时对象，其值与 lhs 一样。这个临时对象用来与 rhs 一起调用 operator+= ，操作的结果被从 operator+返回。这个代码好像不用写得这么隐密。这样写不是更好么？

```
template<class T>
const T operator+(const T& lhs, const T& rhs)
{
  T result(lhs);                     // 拷贝 lhs 到 result 中
  return result += rhs;              // rhs 与它相加并返回结果
}
```

这个模板几乎与前面的程序相同，但是它们之间还是存在重要的差别。第二个模板包含一个命名对象，result。这个命名对象意味着不能在 operator+ 里使用返回值优化（参见条款 M20）。第一种实现方法总可以使用返回值优化，所以编译器为其生成优化代码的可能就会更大。

广告中的事实迫使我指出表达式：

```
return T(lhs) += rhs;
```

比大多数编译器希望进行的返回值优化更复杂。上面第一个函数实现也有这样的临时对象开销，就象你为使用命名对象 result 而耗费的开销一样。然而未命名的对象在历史上比命名对象更容易清除，因此当我们面对在命名对象和临时对象间进行选择时，用临时对象更好一些。它使你耗费的开销不会比命名的对象还多，特别是使用老编译器时，它的耗费会更少。

这里谈论的命名对象、未命名对象和编译优化是很有趣的，但是主要的一点是 operator 的赋值形式（operator+=）比单独形式(operator+)效率更高。做为一个库程序设计者，应该两者都提供，做为一个应用程序的开发者，在优先考虑性能时你应该考虑考虑用 operator 赋值形式代替单独形式。

6.8 Item M23：考虑变更程序库

程序库的设计就是一个折衷的过程。理想的程序库应该是短小的、快速的、强大的、灵活的、可扩展的、直观的、普遍适用的、具有良好的支持、没有使用约束、没有错误的。这也是不存在的。为尺寸和速度而进行优化的程序库一般不能被移植。具有大量功能的的程序库不会具有直观性。没有错误的程序库在使用范围上会有限制。真实的世界里，你不能拥有每一件东西，总得有付出。

不同的设计者给这些条件赋予了不同的优先级。他们从而在设计中牺牲了不同的东西。因此一般两个提供相同功能的程序库却有着完全不同的性能特征。

例如，考虑 iostream 和 stdio 程序库，对于 C++程序员来说两者都是可以使用的。iostream 程序库与 C 中的 stdio 相比有几个优点（参见 Effective C++）。例如它是类型安全的（type-safe），它是可扩展的。然而在效率方面，iostream 程序库总是不如 stdio，因为 stdio 产生的执行文件与 iostream 产生的执行文件相比尺寸小而且执行速度快。

首先考虑执行速度的问题。要想掌握 iostream 和 stdio 之间的性能差别，一种方法就是用这两个程序库来运行 benchmark 程序。不过你必须记住 benchmark 也会撒谎。不仅很难拿出一组数据能够代表程序或程序库的典型用法，而且就算拿出来也是没用，除非有可靠的方法判断出你或你的客户就是典型的代表。不过 benchmark 还是能在同一个问题的不同解决方法间的比较上提供一些信息，所以尽管完全依靠 benchmark 是愚蠢的，但是忽略它们也是愚蠢的。

让我们测试一个简单的 benchmark 程序，只测试最基本的 I/O 功能。这个程序从标准输入读取 30000 个浮点数，然后把它们以固定的格式写到标准输出里。编译时预处理符号 STDIO 决定是使用 stdio 还是 iostream。如果定义了这个符号，就是用 stdio，否则就使用 iostream 程序库。

```
#ifdef STDIO
```

```
#include <stdio.h>
#else
#include <iostream>
#include <iomanip>
using namespace std;
#endif
const int VALUES = 30000;                // # of values to read/write
int main()
{
  double d;
  for (int n = 1; n <= VALUES; ++n) {
#ifdef STDIO
    scanf("%lf", &d);
    printf("%10.5f", d);
#else
    cin >> d;
    cout  << setw(10)                    // 设定 field 宽度
          << setprecision(5)             // 设置小数位置
          << setiosflags(ios::showpoint) // keep trailing 0s
          << setiosflags(ios::fixed)     // 使用这些设置
          << d;
#endif
    if (n % 5 == 0) {
#ifdef STDIO
      printf("\n");
#else
      cout << '\n';
#endif
    }
  }
  return 0;
}
```

当把正整数的自然对数传给这个程序，它会这样输出：

```
0.00000   0.69315   1.09861   1.38629   1.60944
```

```
1.79176    1.94591    2.07944    2.19722    2.30259
2.39790    2.48491    2.56495    2.63906    2.70805
2.77259    2.83321    2.89037    2.94444    2.99573
3.04452    3.09104    3.13549    3.17805    3.21888
```

这种输出表明了使用 **iostreams** 也能这种也能产生 **fixed-format I/O**。当然，

```
cout  << setw(10)
      << setprecision(5)
      << setiosflags(ios::showpoint)
      << setiosflags(ios::fixed)
      << d;
```

远不如 **printf("%10.5f", d);** 输入方便。但是操作符**<<**既是类型安全（**type-safe**）又可以扩展，而 **printf** 则不具有这两种优点。

我做了几种计算机、操作系统和编译器的不同组合，在其上运行这个程序，在每一种情况下都是使用 **stdio** 的程序运行得较快。优势它仅仅快一些（大约 **20**％），有时则快很多（接近 **200**％），但是我从来没有遇到过一种 **iostream** 的实现和与其相对应的 **stdio** 的实现运行速度一样快。另外，使用 **stdio** 的程序的尺寸比与相应的使用 **iostream** 的程序要小（有时是小得多）。（对于程序现实中的尺寸，这点差异就微不足道了）。

应该注意到 **stdio** 的高效性主要是由其代码实现决定的，所以我已经测试过的系统其将来的实现或者我没有测试过的系统的当前实现都可能表现出 **iostream** 和 **stdio** 并没有显著的差异。事实上，有理由相信会发现一种 **iostream** 的代码实现比 **stdio** 要快，因为 **iostream** 在编译时确定它们操作数的类型，而 **stdio** 的函数则是在运行时去解析格式字符串（**format string**）。**iostream** 和 **stdio** 之间性能的对比不过是一个例子，这并不重要，重要的是具有相同功能的不同的程序库在性能上采取不同的权衡措施，所以一旦你找到软件的瓶颈（通过进行 **profile** 参见条款 **M16**），你应该知道是否可能通过替换程序库来消除瓶颈。比如如果你的程序有 **I/O** 瓶颈，你可以考虑用 **stdio** 替代 **iostream**，如果程序在动态分配和释放内存上使用了大量时间，你可以想想是否有其他的 **operator new** 和 **operator delete** 的实现可用（参见条款 **M8** 和 **Effective C++**条款 **10**）。因为不同的程序库在效率、可扩展性、移植性、类型安全和其他一些领域上蕴含着不同的设计理念，通过变换使用给予性能更多考虑的程序库，你有时可以大幅度地提高软件的效率。

6.9 Item M24：理解虚拟函数、多继承、虚基类和 RTTI 所需的代价

C++编译器们必须实现语言的每一个特性。这些实现的细节当然是由编译器来决定的，并且不同的编译器有不同的方法实现语言的特性。在多数情况下，你不用关心这些事情。然而有些特性的实现对对象大小和其成员函数执行速度有很大的影响，所以对于这些特性有一

个基本的了解，知道编译器可能在背后做了些什么，就显得很重要。这种特性中最重要的例子是虚拟函数。

当调用一个虚拟函数时，被执行的代码必须与调用函数的对象的动态类型相一致；指向对象的指针或引用的类型是不重要的。编译器如何能够高效地提供这种行为呢？大多数编译器是使用 **virtual table** 和 **virtual table pointers**。**virtual table** 和 **virtual table pointers** 通常被分别地称为 **vtbl** 和 **vptr**。

一个 **vtbl** 通常是一个函数指针数组。（一些编译器使用链表来代替数组，但是基本方法是一样的）在程序中的每个类只要声明了虚函数或继承了虚函数，它就有自己的 **vtbl**，并且类中 **vtbl** 的项目是指向虚函数实现体的指针。例如，如下这个类定义：

```
class C1 {
public:
  C1();
  virtual ~C1();
  virtual void f1();
  virtual int f2(char c) const;
  virtual void f3(const string& s);
  void f4() const;
  ...
};
```

C1 的 **virtual table** 数组看起来如下图所示：

注意非虚函数 **f4** 不在表中，而且 **C1** 的构造函数也不在。非虚函数（包括构造函数，它也被定义为非虚函数）就象普通的 **C** 函数那样被实现，所以有关它们的使用在性能上没有特殊的考虑。

如果有一个 **C2** 类继承自 **C1**，重新定义了它继承的一些虚函数，并加入了它自己的一些虚函数，

```
class C2: public C1 {
public:
  C2();                                  // 非虚函数
  virtual ~C2();                         // 重定义函数
  virtual void f1();                     // 重定义函数
```

```
    virtual void f5(char *str);                    // 新的虚函数
    ...
};
```

它的 **virtual table** 项目指向与对象相适合的函数。这些项目包括指向没有被 **C2** 重定义的 **C1** 虚函数的指针：

这个论述引出了虚函数所需的第一个代价：你必须为每个包含虚函数的类的 **virtual talbe** 留出空间。类的 **vtbl** 的大小与类中声明的虚函数的数量成正比（包括从基类继承的虚函数）。每个类应该只有一个 **virtual table**，所以 **virtual table** 所需的空间不会太大，但是如果你有大量的类或者在每个类中有大量的虚函数，你会发现 **vtbl** 会占用大量的地址空间。

因为在程序里每个类只需要一个 **vtbl** 拷贝，所以编译器肯定会遇到一个棘手的问题：把它放在哪里。大多数程序和程序库由多个 **object**（目标）文件连接而成，但是每个 **object** 文件之间是独立的。哪个 **object** 文件应该包含给定类的 **vtbl** 呢？你可能会认为放在包含 **main** 函数的 **object** 文件里，但是程序库没有 **main**，而且无论如何包含 **main** 的源文件不会涉及很多需要 **vtbl** 的类。编译器如何知道它们被要求建立那一个 **vtbl** 呢？

必须采取一种不同的方法，编译器厂商为此分成两个阵营。对于提供集成开发环境（包含编译程序和连接程序）的厂商，一种干脆的方法是为每一个可能需要 **vtbl** 的 **object** 文件生成一个 **vtbl** 拷贝。连接程序然后去除重复的拷贝，在最后的可执行文件或程序库里就为每个 **vtbl** 保留一个实例。

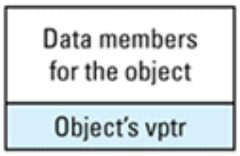
更普通的设计方法是采用启发式算法来决定哪一个 **object** 文件应该包含类的 **vtbl**。通常启发式算法是这样的：要在一个 **object** 文件中生成一个类的 **vtbl**，要求该 **object** 文件包含该类的第一个非内联、非纯虚拟函数（**non-inline non-pure virual function**）定义（也就是类的实现体）。因此上述 **C1** 类的 **vtbl** 将被放置到包含 **C1::~C1** 定义的 **object** 文件里（不是内联的函数），**C2** 类的 **vtbl** 被放置到包含 **C1::~C2** 定义的 **object** 文件里（不是内联函数）。

实际当中，这种启发式算法效果很好。但是如果你过分喜欢声明虚函数为内联函数（参见 **Effective C++**条款 **33**），如果在类中的所有虚函数都内声明为内联函数，启发式算法就会失败，大多数基于启发式算法的编译器会在每个使用它的 **object** 文件中生成一个类的

vtbl。在大型系统里，这会导致程序包含同一个类的成百上千个 **vtbl** 拷贝！大多数遵循这种启发式算法的编译器会给你一些方法来人工控制 **vtbl** 的生成，但是一种更好的解决此问题的方法是避免把虚函数声明为内联函数。下面我们将看到，有一些原因导致现在的编译器一般总是忽略虚函数的的 **inline** 指令。

Virtual table 只实现了虚拟函数的一半机制，如果只有这些是没有用的。只有用某种方法指出每个对象对应的 **vtbl** 时，它们才能使用。这是 **virtual table pointer** 的工作，它来建立这种联系。

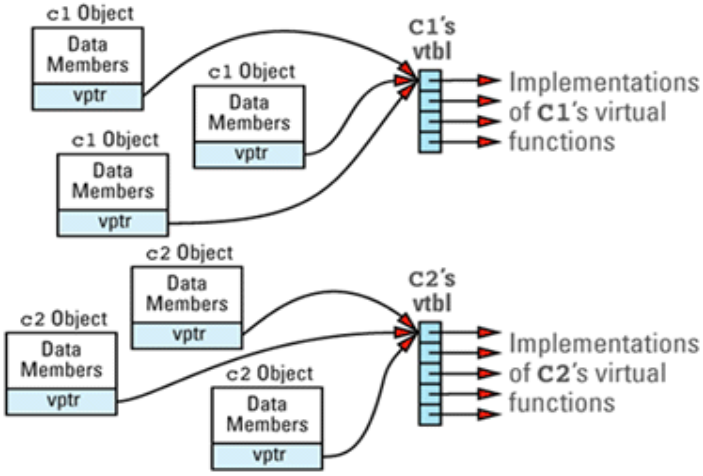
每个声明了虚函数的对象都带有它，它是一个看不见的数据成员，指向对应类的 **virtual table**。这个看不见的数据成员也称为 **vptr**，被编译器加在对象里，位置只有才编译器知道。从理论上讲，我们可以认为包含有虚函数的对象的布局是这样的：



这幅图片表示 **vptr** 位于对象的底部，但是不要被它欺骗，不同的编译器放置它的位置也不同。存在继承的情况下，一个对象的 **vptr** 经常被数据成员所包围。如果存在多继承 **(Multiple inheritance)**，这幅图片会变得更复杂，等会儿我们将讨论它。现在只需简单地记住虚函数所需的第二个代价是：在每个包含虚函数的类的对象里，你必须为额外的指针付出代价。

如果对象很小，这是一个很大的代价。比如如果你的对象平均只有 **4** 比特的成员数据，那么额外的 **vptr** 会使成员数据大小增加一倍（假设 **vptr** 大小为 **4** 比特）。在内存受到限制的系统里，这意味着你必须减少建立对象的数量。即使在内存没有限制的系统里，你也会发现这会降低软件的性能，因为较大的对象有可能不适合放在缓存（**cache**）或虚拟内存页中 **(virtual memory page)**，这就可能使得系统换页操作增多。

假如我们有一个程序，包含几个 **C1** 和 **C2** 对象。对象、**vptr** 和刚才我们讲述的 **vtbl** 之间的关系，在程序里我们可以这样去想象：



考虑这段这段程序代码：

```
void makeACall(C1 *pC1)
{
  pC1->f1();
}
```

通过指针 **pC1** 调用虚拟函数 **f1**。仅仅看这段代码，你不会知道它调用的是那一个 **f1** 函数——**C1::f1** 或 **C2::f1**，因为 **pC1** 可以指向 **C1** 对象也可以指向 **C2** 对象。尽管如此编译器仍然得为在 **makeACall** 的 **f1** 函数的调用生成代码，它必须确保无论 **pC1** 指向什么对象，函数的调用必须正确。编译器生成的代码会做如下这些事情：

1. 通过对象的 **vptr** 找到类的 **vtbl**。这是一个简单的操作，因为编译器知道在对象内哪里能找到 **vptr**（毕竟是由编译器放置的它们）。因此这个代价只是一个偏移调整（以得到 **vptr**）和一个指针的间接寻址（以得到 **vtbl**）。

2. 找到对应 **vtbl** 内的指向被调用函数的指针（在上例中是 **f1**）。这也是很简单的，因为编译器为每个虚函数在 **vtbl** 内分配了一个唯一的索引。这步的代价只是在 **vtbl** 数组内的一个偏移。

3. 调用第二步找到的的指针所指向的函数。

如果我们假设每个对象有一个隐藏的数据叫做 **vptr**，而且 **f1** 在 **vtbl** 中的索引为 **i**，此语句

```
pC1->f1();
```

生成的代码就是这样的

```
(*pC1->vptr[i])(pC1);          //调用被 vtbl 中第 i 个单元指
                               // 向的函数，而 pC1->vptr
                               //指向的是 vtbl；pC1 被做为
                               // this 指针传递给函数。
```

这几乎与调用非虚函数效率一样。在大多数计算机上它多执行了很少的一些指令。调用虚函数所需的代价基本上与通过函数指针调用函数一样。虚函数本身通常不是性能的瓶颈。

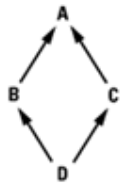
在实际运行中，虚函数所需的代价与内联函数有关。实际上虚函数不能是内联的。这是因为“内联”是指“在编译期间用被调用的函数体本身来代替函数调用的指令，”但是虚函数的“虚”是指“直到运行时才能知道要调用的是哪一个函数。”如果编译器在某个函数的调用点不知道具体是哪个函数被调用，你就能知道为什么它不会内联该函数的调用。这是虚函数所需的第三个代价：你实际上放弃了使用内联函数。（当通过对象调用虚函数时，它可以被内联，但是大多数虚函数是通过对象的指针或引用被调用的，这种调用不能被内联。因为这种调用是标准的调用方式，所以虚函数实际上不能被内联。）

到现在为止我们讨论的东西适用于单继承和多继承，但是多继承的引入，事情就会变得更加复杂（参见 **Effective C++**条款 **43**）。这里详细论述其细节，但是在多继承里，在对象里为寻找 **vptr** 而进行的偏移量计算会变得更复杂。在单个对象里有多个 **vptr**（每一个基类对应一个）;除了我们已经讨论过的单独的自己的 **vtbl** 以外，还得为基类生成特殊的 **vtbl**。因此增加了每个类和每个对象中的虚函数额外占用的空间，而且运行时调用所需的代价也增加了一些。

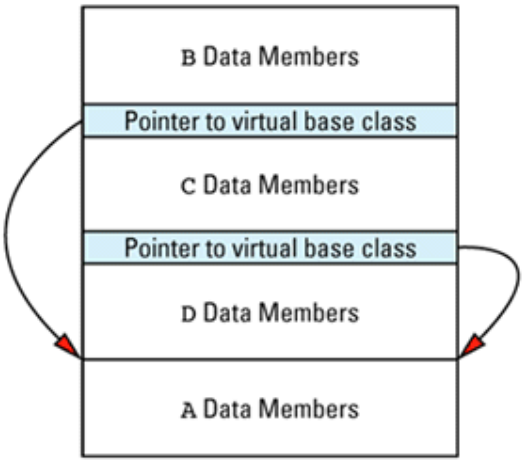
多继承经常导致对虚基类的需求。没有虚基类，如果一个派生类有一个以上从基类的继承路径，基类的数据成员被复制到每一个继承类对象里，继承类与基类间的每条路径都有一个拷贝。程序员一般不会希望发生这种复制，而把基类定义为虚基类则可以消除这种复制。然而虚基类本身会引起它们自己的代价，因为虚基类的实现经常使用指向虚基类的指针做为避免复制的手段，一个或者更多的指针被存储在对象里。

例如考虑下面这幅图，我经常称它为“恐怖的多继承菱形”（**the dreaded multiple inheritance diamond**）

```
class A { ... };
class B: virtual public A { ... };
class C: virtual public A { ... };
class D: public B, public C { ... };
```

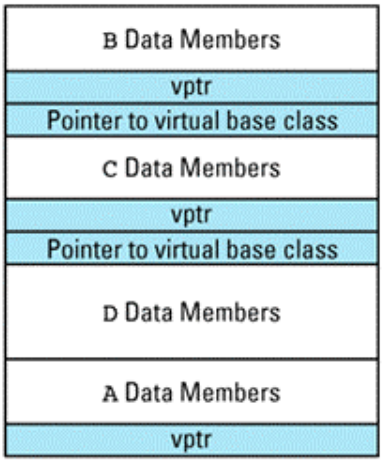


这里 **A** 是一个虚基类，因为 **B** 和 **C** 虚拟继承了它。使用一些编译器（特别是比较老的编译器），**D** 对象会产生这样布局：



把基类的数据成员放在对象的最底端，这显得有些奇怪，但是它经常这么做。当然如何实现是编译器的自由，它们想怎么做都可以，这幅图只是虚基类如何导致对象需要额外指针的概念性描述，所以你不应该在此范围以外还使用这幅图。一些编译器可能加入更少的指针，还有一些编译器会使用某种方法而根本不加入额外的指针（这种编译器让 **vptr** 和 **vtbl** 负担双重责任）。

如果我们把这幅图与前面展示如何把 **virtual table pointer** 加入到对象里的图片合并起来，我们就会认识到如果在上述继承体系里的基类 **A** 有任何虚函数，对象 **D** 的内存布局就是这样的：



这里对象中被编译器加入的部分，我已经做了阴影处理。这幅图可能会有误导，因为阴影部分与非阴影部分之间的面积比例由类中数据量决定。对于小类，额外的代价就大。对于包含更多数据的类，相对来说额外的代价就不大，尽管也是值得注意的。

还有一点奇怪的是虽然存在四个类，但是上述图表只有三个 **vptr**。只要编译器喜欢，当然可以生成四个 **vptr**，但是三个已经足够了（它发现 **B** 和 **D** 能够共享一个 **vptr**），大多数编译器会利用这个机会来减少编译器生成的额外负担。

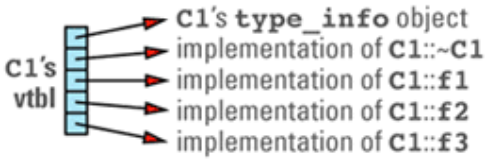
我们现在已经看到虚函数能使对象变得更大，而且不能使用内联，我们已经测试过多继承和虚基类也会增加对象的大小。

让我们转向最后一个话题，运行时类型识别（**RTTI**）。

RTTI 能让我们在运行时找到对象和类的有关信息，所以肯定有某个地方存储了这些信息让我们查询。这些信息被存储在类型为 **type_info** 的对象里，你能通过使用 **typeid** 操作符访问一个类的 **type_info** 对象。

在每个类中仅仅需要一个 **RTTI** 的拷贝，但是必须有办法得到任何对象的类型信息。实际上这叙述得不是很准确。语言规范上这样描述：我们保证可以获得一个对象动态类型信息，如果该类型有至少一个虚函数。这使得 **RTTI** 数据似乎有些象 **virtual function talbe(**虚函数表**)**。每个类我们只需要信息的一个拷贝，我们需要一种方法从任何包含虚函数的对象里获得合适的信息。这种 **RTTI** 和 **virtual function table** 之间的相似点并不是巧合：**RTTI** 被设计为在类的 **vtbl** 基础上实现。

例如，**vtbl** 数组的索引 **0** 处可以包含一个 **type_info** 对象的指针，这个对象属于该 **vtbl** 相对应的类。上述 **C1** 类的 **vtbl** 看上去象这样：



使用这种实现方法，RTTI 耗费的空间是在每个类的 vtbl 中的占用的额外单元再加上存储 type_info 对象的空间。就象在多数程序里 virtual table 所占的内存空间并不值得注意一样，你也不太可能因为 type_info 对象大小而遇到问题。

下面这个表各是对虚函数、多继承、虚基类以及 RTTI 所需主要代价的总结：

Feature	Increases Size of Objects	Increases Per-Class Data	Reduces Inlining
Virtual Functions	Yes	Yes	Yes
Multiple Inheritance	Yes	Yes	No
Virtual Base Classes	Often	Sometimes	No
RTTI	No	Yes	No

一些人看到这个表格以后，会很吃惊，他们宣布“我还是应该使用 C”。很好。但是请记住如果没有这些特性所提供的功能，你必须手工编码来实现。在多数情况下，你的人工模拟可能比编译器生成的代码效率更低，稳定性更差。例如使用嵌套的 switch 语句或层叠的 if－then－else 语句模拟虚函数的调用，其产生的代码比虚函数的调用还要多，而且代码运行速度也更慢。再有，你必须自己人工跟踪对象类型，这意味着对象会携带它们自己的类型标签（type tag）。因此你不会得到更小的对象。

理解虚函数、多继承、虚基类、RTTI 所需的代价是重要的，但是如果你需要这些功能，不管采取什么样的方法你都得为此付出代价，理解这点也同样重要。有时你确实有一些合理的原因要绕过编译器生成的服务。例如隐藏的 vptr 和指向虚基类的指针会使得在数据库中存储 C++对象或跨进程移动它们变得困难，所以你可能希望用某种方法模拟这些特性，能更加容易地完成这些任务。不过从效率的观点来看，你自己编写代码不可能做得比编译器生成的代码更好。

7. 技巧（Techniques，又称 Idioms 或 Pattern）

本书涉及的大多数内容都是编程的指导准则。这些准则虽是重要的，但是程序员不能单靠准则生活。有一个很早以前的卡通片叫做“菲利猫”（Felix the Cat），菲利猫无论何时遇到困难，它都会拿它的 trick 包。如果一个卡通角色都有一个 trick 包，那么 C++程序员就更应该有了。把这一章想成你的 trick 包的启动器。

当设计 C++软件时，总会再三地受到一些问题的困扰。你如何让构造函数和非成员函数具有虚拟函数的特点？你如何限制一个类的实例的数量？你如何防止在堆中建立对象呢？你如何又能确保把对象建立在堆中呢？其它一些类的成员函数无论何时被调用，你如何能建立一个对象并让它自动地完成一些操作？你如何能让不同的对象共享数据结构，而让每个使

用者以为它们每一个都拥有自己的拷贝？你如何区分 **operator[]**的读操作和写操作？你如何建立一个虚函数，其行为特性依赖于不同对象的动态类型？

所有这些问题（还有更多）都在本章得到解答，在本章里我叙述的都是 **C++**程序员普遍遇到的问题，且解决方法也是已被证实了的。我把这些解决方法叫做技巧，不过当它们以程式化的风格（**stylized fashion**）被归档时，也被做为 **idiom** 和 **pattern**。不管你把它称做什么，在你日复一日地从事软件开发工作时，下面这些信息都将使你受益。它会使你相信无论你做什么，总可以用 **C++**来完成它。

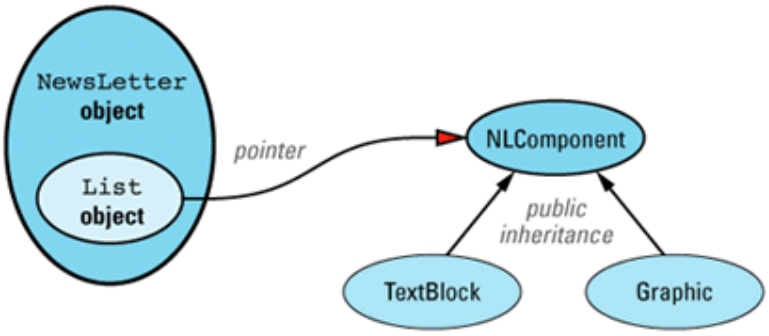
7.1 Item M25：将构造函数和非成员函数虚拟化

从字面来看，谈论“虚拟构造函数”没有意义。当你有一个指针或引用，但是不知道其指向对象的真实类型是什么时，你可以调用虚拟函数来完成特定类型（**type-specific**）对象的行为。仅当你还没拥有一个对象但是你又确切地知道想要的对象的类型时，你才会调用构造函数。那么虚拟构造函数又从何谈起呢？

很简单。尽管虚拟构造函数看起来好像没有意义，其实它们有非常大的用处（如果你认为没有意义的想法就没有用处，那么你怎么解释现代物理学的成就呢？）（因为现代物理学的主要成就是狭义、广义相对论，量子力学，这些理论看起来都好象很荒谬，不好理解。译者注）。例如，假设你编写一个程序，用来进行新闻报道的工作，每一条新闻报道都由文字或图片组成。你可以这样管理它们：

```
class NLComponent {                  //用于 newsletter components
public:                              // 的抽象基类
  ...                                //包含至少一个纯虚函数
};
class TextBlock: public NLComponent {
public:
  ...                                // 不包含纯虚函数
};
class Graphic: public NLComponent {
public:
  ...                                // 不包含纯虚函数
};
class NewsLetter {                   // 一个 newsletter 对象
public:                              // 由 NLComponent 对象
  ...                                // 的链表组成
private:
  list<NLComponent*> components;
};
```

类之间的关系图



在 NewsLetter 中使用的 list 类是一个标准模板类（STL），STL 是标准 C++类库的一部分（参见 Effective C++条款 49 和条款 M35）。list 类型对象的行为特性有些象双向链表，尽管它没有以这种方法来实现。对象 NewLetter 不运行时就会存储在磁盘上。为了能够通过位于磁盘的替代物来建立 Newsletter 对象，让 NewLetter 的构造函数带有 istream 参数是一种很方便的方法。当构造函数需要一些核心的数据结构时，它就从流中读取信息：

```
class NewsLetter {
public:
  NewsLetter(istream& str);
  ...
};
```

此构造函数的伪代码是这样的：

```
NewsLetter::NewsLetter(istream& str)
{
  while (str) {
    从 str 读取下一个 component 对象;
    把对象加入到 newsletter 的 components 对象的链表中去;
  }
}
```

或者，把这种技巧用于另一个独立出来的函数叫做 readComponent，如下所示：

```
class NewsLetter {
public:
  ...
private:
  // 为建立下一个 NLComponent 对象从 str 读取数据,
  // 建立 component 并返回一个指针。
  static NLComponent * readComponent(istream& str);
   ...
};
  NewsLetter::NewsLetter(istream& str)
  {
    while (str) {
        // 把 readComponent 返回的指针添加到 components 链表的最后，
        // "push_back" 一个链表的成员函数，用来在链表最后进行插入操作。
        components.push_back(readComponent(str));
    }
 }
```

考虑一下 readComponent 所做的工作。它根据所读取的数据建立了一个新对象，或是 TextBlock 或是 Graphic。因为它能建立新对象，它的行为与构造函数相似，而且因为它能建立不同类型的对象，我们称它为虚拟构造函数。虚拟构造函数是指能够根据输入给它的数据的不同而建立不同类型的对象。虚拟构造函数在很多场合下都有用处，从磁盘（或者通过网络连接，或者从磁带机上）读取对象信息只是其中的一个应用。（WQ 加注：readComponent() 的实现可详见《汤姆.斯旺 C＋＋编程秘诀》）

还有一种特殊种类的虚拟构造函数－－虚拟拷贝构造函数－－也有着广泛的用途。虚拟拷贝构造函数能返回一个指针，指向调用该函数的对象的新拷贝。因为这种行为特性，虚拟拷贝构造函数的名字一般都是 copySelf，cloneSelf 或者是象下面这样就叫做 clone。很少会有函数能以这么直接的方式实现它：

```
class NLComponent {
public:
  // declaration of virtual copy constructor
  virtual NLComponent * clone() const = 0;
  ...
```

```
};
class TextBlock: public NLComponent {
public:
  virtual TextBlock * clone() const         // virtual copy
  { return new TextBlock(*this); }          // constructor
  ...
};
class Graphic: public NLComponent {
public:
  virtual Graphic * clone() const            // virtual copy
  { return new Graphic(*this); }             // constructor
  ...
};
```

正如我们看到的，类的虚拟拷贝构造函数只是调用它们真正的拷贝构造函数。因此“拷贝”的含义与真正的拷贝构造函数相同。如果真正的拷贝构造函数只做了简单的拷贝，那么虚拟拷贝构造函数也做简单的拷贝。如果真正的拷贝构造函数做了全面的拷贝，那么虚拟拷贝构造函数也做全面的拷贝。如果真正的拷贝构造函数做一些奇特的事情，象引用计数或**copy-on-write**（参见条款**M29**），那么虚拟构造函数也这么做。完全一致，太棒了。

注意上述代码的实现利用了最近才被采纳的较宽松的虚拟函数返回值类型规则。被派生类重定义的虚拟函数不用必须与基类的虚拟函数具有一样的返回类型。如果函数的返回类型是一个指向基类的指针（或一个引用），那么派生类的函数可以返回一个指向基类的派生类的指针（或引用）。这不是 **C++**的类型检查上的漏洞，它使得有可能声明象虚拟构造函数这样的函数。这就是为什么 **TextBlock** 的 **clone** 函数能够返回 **TextBlock***和 **Graphic** 的 **clone** 能够返回 **Graphic***的原因，即使 **NLComponent** 的 **clone** 返回值类型为 **NLComponent***。

在**NLComponent**中的虚拟拷贝构造函数能让实现**NewLetter**的**(**正常的**)**拷贝构造函数变得很容易：

```
class NewsLetter {
public:
  NewsLetter(const NewsLetter& rhs);
  ...
private:
  list<NLComponent*> components;
};
NewsLetter::NewsLetter(const NewsLetter& rhs)
{
  // 遍历整个 rhs 链表，使用每个元素的虚拟拷贝构造函数
  // 把元素拷贝进这个对象的 component 链表。
  // 有关下面代码如何运行的详细情况，请参见条款 M35.
  for (list<NLComponent*>::const_iterator it =
         rhs.components.begin();
       it != rhs.components.end();
       ++it) {
  // "it" 指向 rhs.components 的当前元素，调用元素的 clone 函数，
  // 得到该元素的一个拷贝，并把该拷贝放到
  // 这个对象的 component 链表的尾端。
    components.push_back((*it)->clone());
  }
}
```

如果你对标准模板库（**STL**）不熟悉，这段代码可能有些令人费解，不过原理很简单：遍历被拷贝的 **NewsLetter** 对象中的整个 **component** 链表，调用链表内每个元素对象的虚拟构造函数。我们在这里需要一个虚拟构造函数，因为链表中包含指向 **NLComponent** 对象的指

针，但是我们知道其实每一个指针不是指向 **TextBlock** 对象就是指向 **Graphic** 对象。无论它指向谁，我们都想进行正确的拷贝操作，虚拟构造函数能够为我们做到这点。

虚拟化非成员函数

就象构造函数不能真的成为虚拟函数一样，非成员函数也不能成为真正的虚拟函数（参见 **Effective C++** 条款 **19**）。然而，既然一个函数能够构造出不同类型的新对象是可以理解的，那么同样也存在这样的非成员函数，可以根据参数的不同动态类型而其行为特性也不同。例如，假设你想为 **TextBlock** 和 **Graphic** 对象实现一个输出操作符。显而易见的方法是虚拟化这个输出操作符。但是输出操作符是 **operator<<**，函数把 **ostream&**做为它的左参数（**left-hand argument**）（即把它放在函数参数列表的左边　译者注），这就不可能使该函数成为 **TextBlock** 或 **Graphic** 成员函数。

（这样做也可以，不过看一看会发生什么：

```
class NLComponent {
public:
  // 对输出操作符的不寻常的声明
  virtual ostream& operator<<(ostream& str) const = 0;
  ...
};
class TextBlock: public NLComponent {
public:
  // 虚拟输出操作符(同样不寻常)
  virtual ostream& operator<<(ostream& str) const;
};
class Graphic: public NLComponent {
public:
  // 虚拟输出操作符 (让就不寻常)
  virtual ostream& operator<<(ostream& str) const;
};
TextBlock t;
Graphic g;
...
t << cout;                                  // 通过 virtual operator<<
                                            //把 t 打印到 cout 中。
                                            // 不寻常的语法
g << cout;                                  //通过 virtual operator<<
                                            //把 g 打印到 cout 中。
                                            //不寻常的语法
```

类的使用者得把 **stream** 对象放到**<<**符号的右边，这与输出操作符一般的用发相反。为了能够回到正常的语法上来，我们必须把 **operator<<**移出 **TextBlock** 和 **Graphic** 类，但是如果我们这样做，就不能再把它声明为虚拟了。）

另一种方法是为打印操作声明一个虚拟函数（例如 **print**）把它定义在 **TextBlock** 和 **Graphic** 类里。但是如果这样，打印 **TextBlock** 和 **Graphic** 对象的语法就与使用 **operator<<** 做为输出操作符的其它类型的对象不一致了，这些解决方法都不很令人满意。我们想要的是一个称为 **operator<<**的非成员函数，其具有象 **print** 虚拟函数的行为特性。有关我们想要什么的描述实际上已经很接近如何得到它的描述。我们定义 **operator<<** 和 **print** 函数，让前者调用后者！

```
class NLComponent {
public:
  virtual ostream& print(ostream& s) const = 0;
  ...
};
class TextBlock: public NLComponent {
```

```
public:
  virtual ostream& print(ostream& s) const;
  ...
};
class Graphic: public NLComponent {
public:
  virtual ostream& print(ostream& s) const;
  ...
};
inline
ostream& operator<<(ostream& s, const NLComponent& c)
{
  return c.print(s);
}
```

具有虚拟行为的非成员函数很简单。你编写一个虚拟函数来完成工作，然后再写一个非虚拟函数，它什么也不做只是调用这个虚拟函数。为了避免这个句法花招引起函数调用开销，你当然可以内联这个非虚拟函数（参见 **Effective C++** 条款 **33**）。

现在你知道如何根据它们的一个参数让非成员函数虚拟化，你可能想知道是否可能让它们根据一个以上的参数虚拟化呢？可以，但是不是很容易。有多困难呢？参见条款 **M31**；它将专门论述这个问题。

7.2 Item M26：限制某个类所能产生的对象数量

你很痴迷于对象，但是有时你又想束缚住你的疯狂。例如你在系统中只有一台打印机，所以你想用某种方式把打印机对象数目限定为一个。或者你仅仅取得 **16** 个可分发出去的文件描述符，所以应该确保文件描述符对象存在的数目不能超过 **16** 个。你如何能够做到这些呢？如何去限制对象的数量呢？

如果这是一个用数学归纳法进行的证明，你会从 **n=1** 开始证明，然后从此出发推导出其它证明。幸运的是这既不是一个证明也不是一个归纳。而从 **n=0** 开始更具有启发性，所以我们就从这里开始。你如何能够彻底阻止对象实例化（**instantiate**）呢？

允许建立零个或一个对象

每次实例化一个对象时，我们很确切地知道一件事情：“将调用一个构造函数。”事实确实这样，阻止建立某个类的对象，最容易的方法就是把该类的构造函数声明在类的 **private** 域：

```
class CantBeInstantiated {
private:
  CantBeInstantiated();
  CantBeInstantiated(const CantBeInstantiated&);
...
};
```

这样做以后，每个人都没有权力建立对象，我们能够有选择性地放松这个限制。例如如果想为打印机建立类，但是要遵守我们只有一个对象可用的约束，我们应把打印机对象封装

在一个函数内，以便让每个人都能访问打印机，但是只有一个打印机对象被建立。：

```
class PrintJob;                                  // forward 声明
                                                 // 参见 Effective C++条款 34
class Printer {
public:
  void submitJob(const PrintJob& job);
  void reset();
  void performSelfTest();
...
friend Printer& thePrinter();
private:
  Printer();
  Printer(const Printer& rhs);
  ...
};
Printer& thePrinter()
{
  static Printer p;                              // 单个打印机对象
  return p;
}
```

这个设计由三个部分组成，第一、**Printer** 类的构造函数是 **private**。这样能阻止建立对象。第二、全局函数 **thePrinter** 被声明为类的友元，让 **thePrinter** 避免私有构造函数引起的限制。最后 **thePrinter** 包含一个静态 **Printer** 对象，这意味着只有一个对象被建立。客户端代码无论何时要与系统的打印机进行交互访问，它都要使用 **thePrinter** 函数：

```
class PrintJob {
public:
  PrintJob(const string& whatToPrint);
  ...
};
string buffer;
...                                              //填充 buffer
thePrinter().reset();
thePrinter().submitJob(buffer);
```

当然你感到 **thePrinter** 使用全局命名空间完全是多余的。“是的”，你会说，“全局函

数看起来象全局变量，但是全局变量是 gauche(笨拙的)，我想把所有与打印有关的功能都放到 Printer 类里。”好的，我绝不敢与使用象 gauche 这样的词的人争论（Lippman 的一贯幽默），这很简单，只要在 Prihter 类中声明 thePrinter 为静态函数，然后把它放在你想放的位置。就不再需要友元声明了。使用静态函数，如下所示：

```
class Printer {
public:
  static Printer& thePrinter();
  ...
private:
  Printer();
  Printer(const Printer& rhs);
  ...
};
Printer& Printer::thePrinter()
{
  static Printer p;
  return p;
}
```

用户使用 printer 时有些繁琐：

```
Printer::thePrinter().reset();
Printer::thePrinter().submitJob(buffer);
```

另一种方法是把 thePrinter 移出全局域，放入 namespace（命名空间）（参见 Effective C++条款 28）。命名空间是 C++一个较新的特性。任何能在全局域声明东西也能在命名空间里声明。包括类、结构、函数、变量、对象、typedef 等等。把它们放入命名空间并不影响它们的行为特性，不过能够防止在不同命名空间里的实体发生命名冲突。把 Printer 类和 thePrinter 函数放入一个命名空间，我们就不用担心别人也会使用 Printer 和 thePrinter 名字；命名空间能够防止命名冲突。

命名空间从句法上来看有些象类，但是它没有 public、protected 或 private 域。所有都是 public。如下所示，我们把 Printer、thePrinter 放入叫做 PrintingStuff 的命名空间里：

```
namespace PrintingStuff {
  class Printer {                          // 在命名空间
  public:                                  // PrintingStuff 中的类
    void submitJob(const PrintJob& job);
```

```
    void reset();
    void performSelfTest();
    ...
    friend Printer& thePrinter();
  private:
    Printer();
    Printer(const Printer& rhs);
    ...
 };
 Printer& thePrinter()                    // 这个函数也在命名空间里
 {
    static Printer p;
    return p;
 }
}
                                          // 命名空间到此结束
```

使用这个命名空间后，客户端可以通过使用 **fully-qualified name**（完全限制符名）（即包括命名空间的名字），

```
PrintingStuff::thePrinter().reset();
PrintingStuff::thePrinter().submitJob(buffer);
```

但是也可以使用 **using** 声明，以简化键盘输入：

```
using PrintingStuff::thePrinter;     // 从命名空间"PrintingStuff"
                                     //引入名字"thePrinter"
                                      // 使其成为当前域
thePrinter().reset();                // 现在可以象使用局部命名
thePrinter().submitJob(buffer);      // 一样，使用 thePrinter
```

在 **thePrinter** 的实现上有两个微妙的不引人注目的地方，值得我们看一看。第一，唯一的 **Pritner** 对象是位于函数里的静态成员而不是在类中的静态成员，这样做是非常重要的。在类中的静态对象实际上总是被构造（和释放），即使不使用该对象。与此相反，只有第一次执行函数时，才会建立函数中的静态对象，所以如果没有调用函数，就不会建立对象。（不过你得为此付出代价，每次调用函数时都得检查是否需要建立对象。）建立 **C++**一个理论支柱是你不需为你不用的东西而付出，在函数里，把类似于 **Printer** 这样的对象定义为静态成员就是坚持这样的理论。你应该尽可能坚持这种理论。

与一个函数的静态成员相比，把 **Printer** 声明为类中的静态成员还有一个缺点，它的初

始化时间不确定。我们能够准确地知道函数的静态成员什么时候被初始化：“在第一次执行定义静态成员的函数时”。而没有定义一个类的静态成员被初始化的时间。C++为一个translation unit（也就是生成一个object文件的源代码的集合）内的静态成员的初始化顺序提供某种保证，但是对于在不同translation unit中的静态成员的初始化顺序则没有这种保证（参见Effective C++条款47）。在实际使用中，这会给我们带来许多麻烦。当函数的静态成员能够满足我们的需要时，我们就能避免这些麻烦。在这里的例子里，既然它能够满足需要，我们为什么不用它呢？

第二个细微之处是内联与函数内静态对象的关系。再看一下thePrinter的非成员函数形式：

```
Printer& thePrinter()
{
  static Printer p;
  return p;
}
```

除了第一次执行这个函数时（也就是构造p时），其它时候这就是一个一行函数——它由“return p;”一条语句组成。这个函数最适合做为内联函数使用。然而它不能被声明为内联。为什么呢？请想一想，为什么你要把对象声明为静态呢？通常是因为你只想要该对象的一个拷贝。现在再考虑“内联”意味着什么呢？从概念上讲，它意味着编译器用函数体替代该对函数的每一个调用，不过非成员函数还不只这些。非成员函数还有其它的含义。它还意味着internal linkage（内部链接）。

通常情况下，你不需要理解这种语言上令人迷惑的东西，你只需记住一件事：“带有内部链接的函数可能在程序内被复制（也就是说程序的目标（object）代码可能包含一个以上的内部链接函数的代码），这种复制也包括函数内的静态对象。”结果如何？如果建立一个包含局部静态对象的非成员函数，你可能会使程序的静态对象的拷贝超过一个！所以不要建立包含局部静态数据的非成员函数。

但是你可能认为建立函数来返回一个隐藏对象的引用，这种限制对象的数量的方法是错误的。也许你认为只需简单地计算对象的数目，一旦需要太多的对象，就抛出异常，这样做也许会更好。如下所示，这样建立printer对象，：

```
class Printer {
public:
  class TooManyObjects{};                    // 当需要的对象过多时
                                             // 就使用这个异常类
  Printer();
  ~Printer();
```

```
  ...
private:
  static size_t numObjects;
  Printer(const Printer& rhs);          // 这里只能有一个 printer，
                                        // 所以不允许拷贝
};                                      // （参见 Effective C++ 条款 27）
```

此法的核心思想就是使用 **numObjects** 跟踪 **Pritner** 对象存在的数量。当构造类时，它的值就增加，释放类时，它的值就减少。如果试图构造过多的 **Printer** 对象，就会抛出一个 **TooManyObjects** 类型的异常：

```
// Obligatory definition of the class static
size_t Printer::numObjects = 0;
Printer::Printer()
{
  if (numObjects >= 1) {
    throw TooManyObjects();
  }
  继续运行正常的构造函数;
  ++numObjects;
}
Printer::~Printer()
{
  进行正常的析构函数处理;
  --numObjects;
}
```

这种限制建立对象数目的方法有两个较吸引人的优点。一个是它是直观的，每个人都能理解它的用途。另一个是很容易推广它的用途，可以允许建立对象最多的数量不是一，而是其它大于一的数字。

l 建立对象的环境

这种方法也有一个问题。假设我们一个特殊的打印机，是彩色打印机。这种打印机类有许多地方与普通的打印机类相同，所以我们从普通打印类继承下来：

```
class ColorPrinter: public Printer {
  ...

};
```

现在假设我们系统有一个普通打印机和一个彩色打印机：

```
Printer p;
ColorPrinter cp;
```

这两个定义会产生多少 **Pritner** 对象？答案是两个：一个是 **p**，一个是 **cp**。在运行时，当构造 **cp** 的基类部分时，会抛出 **TooManyObjects** 异常。对于许多程序员来说，这可不是他们所期望的事情。（设计时避免从其它的 **concrete** 类继承 **concrete** 类，就不会遇到这种问题。这种设计思想详见条款 **M33**）。

当其它对象包含 **Printer** 对象时，会发生同样的问题：

```
class CPFMachine {                          // 一种机器，可以复印，打印
private:                                    // 发传真。

  Printer p;                                // 有打印能力
  FaxMachine f;                             // 有传真能力
  CopyMachine c;                            // 有复印能力

  ...

};

CPFMachine m1;                              // 运行正常

CPFMachine m2;                              // 抛出 TooManyObjects 异常
```

问题是 **Printer** 对象能存在于三种不同的环境中：只有它们本身；作为其它派生类的基类；被嵌入在更大的对象里。存在这些不同环境极大地混淆了跟踪“存在对象的数目”的含义，因为你心目中的“对象的存在” 的含义与编译器不一致。

通常你仅会对允许对象本身存在的情况感兴趣，你希望限制这种实例（**instantiation**）的数量。如果你使用最初的 **Printer** 类示例的方法，就很容易进行这种限制，因为 **Printer** 构造函数是 **private**，（不存在 **friend** 声明）带有 **private** 构造函数的类不能作为基类使用，也不能嵌入到其它对象中。你不能从带有 **private** 构造函数的类派生出新类，这个事实导致产生了一种阻止派生类的通用方法，这种方法不需要和限制对象实例数量的方法一起使用。例如，你有一个类 **FSA**，表示一个 **finite state automata(**有限态自动机**)** 。（这种机器能用于很多环境下，比如用户界面设计），并假设你允许建立任意数量的对象，但是你想禁止从 **FSA** 派生出新类。（这样做的一个原因是表明在 **FSA** 中存在非虚析构函数。**Effective C++** 条款 **14** 解释了为什么基类通常需要虚拟析构函数，本书条款 **M24** 解释了为什么没有虚函数

的类比同等的具有虚函数的类要小。）如下所示，这样设计 FSA 可以满足你的这两点需求：

```
class FSA {
public:
  // 伪构造函数
  static FSA * makeFSA();
  static FSA * makeFSA(const FSA& rhs);
  ...
private:
  FSA();
  FSA(const FSA& rhs);
  ...
};
FSA * FSA::makeFSA()
{ return new FSA(); }
FSA * FSA::makeFSA(const FSA& rhs)
{ return new FSA(rhs); }
```

不象 thePrinter 函数总是返回一个对象的引用（引用的对象是固定的），每个 makeFSA 的伪构造函数则是返回一个指向对象的指针（指向的对象都是惟一的，不相同的）。也就是说允许建立的 FSA 对象数量没有限制。

那好，不过每个伪构造函数都调用 new 这个事实暗示调用者必须记住调用 delete。否则就会发生资源泄漏。如果调用者希望退出生存空间时 delete 会被自动调用，他可以把 makeFSA 返回的指针存储在 auto_ptr 中（参见条款 M9）；当它们自己退出生存空间时，这种对象能自动地删除它们所指向的对象：

```
// 间接调用缺省 FSA 构造函数
auto_ptr<FSA> pfsa1(FSA::makeFSA());
// indirectly call FSA copy constructor
auto_ptr<FSA> pfsa2(FSA::makeFSA(*pfsa1));
...                          // 象通常的指针一样使用 pfsa1 和 pfsa2,
                             //不过不用操心删除它们。
```

l 允许对象来去自由

我们知道如何设计只允许建立一个实例的类，我们知道跟踪特定类的对象数量的工作是复杂的，因为在三种不同的环境中都可能调用对象的构造函数，我们知道消除对象计数中混乱现象的方法是把构造函数声明为 private。还有最后一点值得我们注意：使用 thePrinter 函数封装对单个对象的访问，以便把 Printer 对象的数量限制为一个，这样做的同时也会让

我们在每一次运行程序时只能使用一个 **Printer** 对象。导致我们不能这样编写代码：

建立 **Printer** 对象 **p1;**

使用 **p1;**

释放 **p1;**

建立 **Printer** 对象 **p2;**

使用 **p2;**

释放 **p2;**

...

这种设计在同一时间里没有实例化多个 **Printer** 对象，而是在程序的不同部分使用了不同的 **Printer** 对象。不允许这样编写有些不合理。毕竟我们没有违反只能存在一个 **printer** 的约束。就没有办法使它合法化么？

当然有。我们必须把先前使用的对象计数的代码与刚才看到的伪构造函数代码合并在一起：

```
class Printer {
public:
  class TooManyObjects{};
  // 伪构造函数
  static Printer * makePrinter();
~Printer();
  void submitJob(const PrintJob& job);
  void reset();
  void performSelfTest();
  ...
private:
  static size_t numObjects;
  Printer();
  Printer(const Printer& rhs);          //我们不定义这个函数
};                                      //因为不允许
                                         //进行拷贝
                                       // (参见 Effective C++条款 27)
// Obligatory definition of class static
size_t Printer::numObjects = 0;
Printer::Printer()
{
```

```
  if (numObjects >= 1) {
    throw TooManyObjects();
  }
  继续运行正常的构造函数;
  ++numObjects;
}
Printer * Printer::makePrinter()
{ return new Printer; }
```

当需要的对象过多时，会抛出异常，如果你认为这种方式给你的感觉是 **unreasonably harsh**，你可以让伪构造函数返回一个空指针。当然用户在使用之前应该进行检测。

除了用户必须调用伪构造函数，而不是真正的构造函数之外，它们使用 **Printer** 类就象使用其他类一样：

```
Printer p1;                          // 错误! 缺省构造函数是 private
Printer *p2 =
  Printer::makePrinter();              // 正确, 间接调用缺省构造函数
Printer p3 = *p2;                    // 错误! 拷贝构造函数是 private
p2->performSelfTest();               // 所有其它的函数都可以
p2->reset();                         // 正常调用
...
delete p2;                           // 避免内存泄漏，如果
                                     // p2 是一个 auto_ptr,
                                     // 就不需要这步。
```

这种技术很容易推广到限制对象为任何数量上。我们只需把 **hard-wired** 常量值 **1** 改为根据某个类而确定的数量，然后消除拷贝对象的约束。例如，下面这个经过修改的 **Printer** 类的代码实现，最多允许 **10** 个 **Printer** 对象存在：

```
class Printer {
public:
  class TooManyObjects{};
  // 伪构造函数
  static Printer * makePrinter();
  static Printer * makePrinter(const Printer& rhs);
  ...
private:
  static size_t numObjects;
```

```
  static const size_t maxObjects = 10;        // 见下面解释
  Printer();
  Printer(const Printer& rhs);
};
// Obligatory definitions of class statics
size_t Printer::numObjects = 0;
const size_t Printer::maxObjects;
Printer::Printer()
{
  if (numObjects >= maxObjects) {
    throw TooManyObjects();
  }
  ...
}
Printer::Printer(const Printer& rhs)
{
  if (numObjects >= maxObjects) {
    throw TooManyObjects();
  }
  ...
}
Printer * Printer::makePrinter()
{ return new Printer; }
Printer * Printer::makePrinter(const Printer& rhs)
{ return new Printer(rhs); }
```

如果你的编译器不能编译上述类中 **Printer::maxObjects** 的声明，这丝毫也不奇怪。特别是应该做好准备，编译器不能编译把 **10** 做为初值赋给这个变量这条语句。给 **static const** 成员（例如 **int, char, enum** 等等）确定初值的功能是最近才加入到 **C++**中的，所以一些编译器还不允许这样编写。如果没有及时更新你的编译器，可以把 **maxObjects** 声明为在一个 **private** 内匿名枚举类型里的枚举元素，

```
class Printer {
private:
  enum { maxObjects = 10 };                // 在类中，
  ...                                      // maxObjects 为常量 10
```

```
};
```

或者象 **non-const static** 成员一样初始化 **static** 常量：

```
class Printer {
private:
  static const size_t maxObjects;              // 没有赋给初值
  ...
};
// 放在一个代码实现的文件中
const size_t Printer::maxObjects = 10;
```

后面这种方法与原来的方法有一样的效果，但是显示地确定初值能让其他程序员更容易理解。当你的编译器支持在类定义中给 **const static** 成员赋初值的功能时，你应该尽可能地利用这个功能。

一个具有对象计数功能的基类

把初始化静态成员撇在一边不说，上述的方法使用起来就像咒语一样灵验，但是另一方面它也有些繁琐。如果我们有大量像 **Printer** 需要限制实例数量的类，就必须一遍又一遍地编写一样的代码，每个类编写一次。这将会使大脑变得麻木。应该有一种方法能够自动处理这些事情。难道没有方法把实例计数的思想封装在一个类里吗？

我们很容易地能够编写一个具有实例计数功能的基类，然后让像 **Printer** 这样的类从该基类继承，而且我们能做得更好。我们使用一种方法封装全部的计数功能，不但封装维护实例计数器的函数，而且也封装实例计数器本身。（当我们在条款 **M29** 中测试引用计数时，将看到需要同样的技术。有关这种设计的测试细节，参见我的文章 **counting objects**。）**Printer** 类的计数器是静态变量 **numObjects**，我们应该把变量放入实例计数类中。然而也需要确保每个进行实例计数的类都有一个相互隔离的计数器。使用计数类模板可以自动生成适当数量的计数器，因为我们能让计数器成为从模板中生成的类的静态成员：

```
template<class BeingCounted>
class Counted {
public:
  class TooManyObjects{};                      // 用来抛出异常
  static int objectCount() { return numObjects; }
protected:
  Counted();
  Counted(const Counted& rhs);
  ~Counted() { --numObjects; }
private:
```

```
  static int numObjects;
  static const size_t maxObjects;
  void init();                                  // 避免构造函数的
};                                              // 代码重复
template<class BeingCounted>
Counted<BeingCounted>::Counted()
{ init(); }
template<class BeingCounted>
Counted<BeingCounted>::Counted(const Counted<BeingCounted>&)
{ init(); }
template<class BeingCounted>
void Counted<BeingCounted>::init()
{
  if (numObjects >= maxObjects) throw TooManyObjects();
  ++numObjects;
}
```

从这个模板生成的类仅仅能被做为基类使用，因此构造函数和析构函数被声明为 **protected**。注意 **private** 成员函数 **init** 用来避免两个 **Counted** 构造函数的语句重复。

现在我们能修改 **Printer** 类，这样使用 **Counted** 模板：

```
class Printer: private Counted<Printer> {
public:
  // 伪构造函数
  static Printer * makePrinter();
  static Printer * makePrinter(const Printer& rhs);
  ~Printer();
  void submitJob(const PrintJob& job);
  void reset();
  void performSelfTest();
  ...
  using Counted<Printer>::objectCount;     // 参见下面解释
  using Counted<Printer>::TooManyObjects;  // 参见下面解释
private:
  Printer();
  Printer(const Printer& rhs);
```

```
};
```

Printer 使用了 Counter 模板来跟踪存在多少 Printer 对象，坦率地说，除了 Printer 的编写者，没有人关心这个事实。它的实现细节最好是 private，这就是为什么这里使用 private 继承的原因（参见 Effective C++条款 42）。另一种方法是在 Printer 和 counted<Printer>之间使用 public 继承，但是我们必须给 Counted 类一个虚拟析构函数。（否则如果有人通过 Counted<Printer>*指针删除一个 Printer 对象，我们就有导致对象行为不正确的风险——参见 Effective C++条款 14。）条款 M24 已经说得很明白了，在 Counted 中存在虚函数，几乎肯定影响从 Counted 继承下来的对象的大小和布局。我们不想引入这些额外的负担，所以使用 private 继承来避免这些负担。

Counted 所做的大部分工作对于 Printer 的用户来说都是隐藏的，但是这些用户可能很想知道有当前多少 Printer 对象存在。Counted 模板提供了 objectCount 函数，用来提供这种信息，但是因为我们使用 private 继承，这个函数在 Printer 类中成为了 private。为了恢复该函数的 public 访问权，我们使用 using 声明：

```
class Printer: private Counted<Printer> {
public:
  ...
  using Counted<Printer>::objectCount; // 让这个函数对于 Printer
                                       //是 public
  ...
};
```

这样做是合乎语法规则的，但是如果你的编译器不支持命名空间，编译器就不允许这样做。如果这样的话，你应使用老式的访问权声明语法：

```
class Printer: private Counted<Printer> {
public:
  ...
  Counted<Printer>::objectCount;       // 让 objectCount
                                       // 在 Printer 中是 public
  ...
};
```

这种更传统的语法与 uning 声明具有相同的含义。但是我们不赞成这样做。TooManyObjects 类应该也应用同样的方式来处理，因为 Printer 的客户端如果要捕获这种异常类型，它们必须有能力访问 TooManyObjects。当 Printer 继承 Counted<Printer>时，它可以忘记有关对象计数的事情。编写 Printer 类时根本不用考虑对象计数，就好像有其他人会为它计数一样。Printer 的构造函数可以是这样的：

```
Printer::Printer()
{
  进行正常的构造函数运行
}
```

这里有趣的不是你所见到的东西，而是你看不到的东西。不检测对象的数量就好像限制将被超过，执行完构造函数后也不增加存在对象的数目。所有这些现在都由**Counted<Printer>**的构造函数来处理，因为 **Counted<Printer>**是 **Printer** 的基类，我们知道 **Counted<Printer>**的构造函数总在 **Printer** 的前面被调用。如果建立过多的对象，**Counted<Printer>**的构造函数就会抛出异常，甚至都没有调用 **Printer** 的构造函数。

最后还有一点需要注意，必须定义 **Counted** 内的静态成员。对于 **numObjects** 来说，这很容易——我们只需要在 **Counted** 的实现文件里定义它即可：

```
template<class BeingCounted>                    // 定义 numObjects
int Counted<BeingCounted>::numObjects;          // 自动把它初始化为 0
```

对于 **maxObjects** 来说，则有一些技巧。我们应该把它初始化为什么值呢？如果你想允许建立 **10** 个 **printer** 对象，我们应该初始化 **Counted<Printer>::maxObjects** 为 **10**。另一方面如果我们向允许建立 **16** 个文件描述符对象，我们应该初始化 **Counted<Printer>::maxObjects** 为 **16**。到底应该怎么做呢？

简单的方法就是什么也不做。我们不对 **maxObject** 进行初始化。而是让此类的客户端提供合适的初始化。**Printer** 的作者必须把这条语句加入到一个实现文件里：

```
const size_t Counted<Printer>::maxObjects = 10;
```

同样 **FileDescriptor** 的作者也得加入这条语句：

```
const size_t Counted<FileDescriptor>::maxObjects = 16;
```

如果这些作者忘了对 **maxObjects** 进行初始化，会发生什么情况呢？很简单：连接时会发生错误，因为 **maxObjects** 没有被定义。如果我们提供了充分的文档对 **Counted** 客户端说明了需求，他们会回去加上这个必须的初始化。

7.3 Item M27：要求或禁止在堆中产生对象

有时你想这样管理某些对象，要让某种类型的对象能够自我销毁，也就是能够“**delete this**”。很明显这种管理方式需要此类型对象被分配在堆中。而其它一些时候你想获得一种保障：“不在堆中分配对象，从而保证某种类型的类不会发生内存泄漏。”如果你在嵌入式系统上工作，就有可能遇到这种情况，发生在嵌入式系统上的内存泄漏是极其严重的，其堆空间是非常珍贵的。有没有可能编写出代码来要求或禁止在堆中产生对象（**heap-based object**）呢？通常是可以的，不过这种代码也会把“**on the heap**”的概念搞得比你脑海中所想的要模糊。

I 要求在堆中建立对象

让我们先从必须在堆中建立对象开始说起。为了执行这种限制，你必须找到一种方法禁止以调用“**new**”以外的其它手段建立对象。这很容易做到。非堆对象（**non-heap object**）

在定义它的地方被自动构造，在生存时间结束时自动被释放，所以只要禁止使用隐式的构造函数和析构函数，就可以实现这种限制。

把这些调用变得不合法的一种最直接的方法是把构造函数和析构函数声明为 **private**。这样做副作用太大。没有理由让这两个函数都是 **private**。最好让析构函数成为 **private**，让构造函数成为 **public**。处理过程与条款 **26** 相似，你可以引进一个专用的伪析构函数，用来访问真正的析构函数。客户端调用伪析构函数释放他们建立的对象。（**WQ** 加注：注意，异常处理体系要求所有在栈中的对象的析构函数必须申明为公有!）

例如，如果我们想仅仅在堆中建立代表 **unlimited precision numbers**（无限精确度数字）的对象，可以这样做：

```
class UPNumber {
public:
  UPNumber();
  UPNumber(int initValue);
  UPNumber(double initValue);
  UPNumber(const UPNumber& rhs);
  // 伪析构函数 (一个 const 成员函数， 因为
  // 即使是 const 对象也能被释放。)
  void destroy() const { delete this; }
  ...
private:
  ~UPNumber();
};
```

然后客户端这样进行程序设计：

```
UPNumber n;                          // 错误! (在这里合法， 但是
                                     // 当它的析构函数被隐式地
                                     // 调用时，就不合法了)
UPNumber *p = new UPNumber;          //正确
...
delete p;                            // 错误! 试图调用
                                     // private 析构函数
p->destroy();                        // 正确
```

另一种方法是把全部的构造函数都声明为 **private**。这种方法的缺点是一个类经常有许多构造函数，类的作者必须记住把它们都声明为 **private**。否则如果这些函数就会由编译器生成，构造函数包括拷贝构造函数，也包括缺省构造函数；编译器生成的函数总是 **public**（参见 **Effecitve C++** 条款 **45**）。因此仅仅声明析构函数为 **private** 是很简单的，因为每个类只有一个析构函数。

通过限制访问一个类的析构函数或它的构造函数来阻止建立非堆对象，但是在条款 **26** 已经说过，这种方法也禁止了继承和包容（**containment**）：

```
class UPNumber { ... };              // 声明析构函数或构造函数
                                     // 为 private
class NonNegativeUPNumber:
  public UPNumber { ... };           // 错误! 析构函数或
                                     //构造函数不能编译
class Asset {
private:
  UPNumber value;
  ...                                // 错误! 析构函数或
                                     //构造函数不能编译
};
```

这些困难不是不能克服的。通过把 **UPNumber** 的析构函数声明为 **protected**（同时它的构造函数还保持 **public**）就可以解决继承的问题，需要包含 **UPNumber** 对象的类可以修改为包

含指向 UPNumber 的指针：

```
class UPNumber { ... };              // 声明析构函数为 protected
class NonNegativeUPNumber:
  public UPNumber { ... };           // 现在正确了；派生类
                                     // 能够访问
                                     // protected 成员
class Asset {
public:
  Asset(int initValue);
  ~Asset();
  ...
private:
  UPNumber *value;
};
Asset::Asset(int initValue)
: value(new UPNumber(initValue))     // 正确
{ ... }
Asset::~Asset()
{ value->destroy(); }                // 也正确
```

l 判断一个对象是否在堆中

如果我们采取这种方法，我们必须重新审视一下“在堆中”这句话的含义。上述粗略的类定义表明一个非堆的 NonNegativeUPNumber 对象是合法的：

```
NonNegativeUPNumber n;               // 正确
```

那么现在 NonNegativeUPNumber 对象 n 中的 UPNumber 部分也不在堆中，这样说对么？答案要依据类的设计和实现的细节而定，但是让我们假设这样说是不对的，所有 UPNumber 对象 —即使是做为其它派生类的基类—也必须在堆中。我们如何能强制执行这种约束呢？

没有简单的办法。UPNumber 的构造函数不可能判断出它是否做为堆对象的基类而被调用。也就是说对于 UPNumber 的构造函数来说没有办法侦测到下面两种环境的区别：

```
NonNegativeUPNumber *n1 =
  new NonNegativeUPNumber;           // 在堆中
NonNegativeUPNumber n2;              //不再堆中
```

不过你可能不相信我。也许你想你能够在 new 操作符、operator new 和 new 操作符调用的构造函数的相互作用中玩些小把戏（参见条款 M8）。可能你认为你比他们都聪明，可以这样修改 UPNumber，如下所示：

```
class UPNumber {
public:
  // 如果建立一个非堆对象，抛出一个异常
  class HeapConstraintViolation {};
  static void * operator new(size_t size);
  UPNumber();
  ...
private:
  static bool onTheHeap;                 //在构造函数内，指示
                                         // 对象是否被构造在
  ...                                    // 堆上
};
// obligatory definition of class static
bool UPNumber::onTheHeap = false;
void *UPNumber::operator new(size_t size)
{
  onTheHeap = true;
```

```
  return ::operator new(size);
}
UPNumber::UPNumber()
{
  if (!onTheHeap) {
    throw HeapConstraintViolation();
  }
  proceed with normal construction here;
  onTheHeap = false;                    // 为下一个对象清除标记
}
```

如果不再深入研究下去，就不会发现什么错误。这种方法利用了这样一个事实："当在堆上分配对象时，会调用 operator new 来分配 raw memory"，operator new 设置 onTheHeap 为 true，每个构造函数都会检测 onTheHeap，看对象的 raw memory 是否被 operator new 所分配。如果没有，一个类型为 HeapConstraintViolation 的异常将被抛出。否则构造函数如通常那样继续运行，当构造函数结束时，onTheHeap 被设置为 false，然后为构造下一个对象而重置到缺省值。

这是一个非常好的方法，但是不能运行。请考虑一下这种可能的客户端代码：

```
UPNumber *numberArray = new UPNumber[100];
```

第一个问题是为数组分配内存的是 operator new[]，而不是 operator new，不过（倘若你的编译器支持它）你能象编写 operator new 一样容易地编写 operator new[]函数。更大的问题是 numberArray 有 100 个元素，所以会调用 100 次构造函数。但是只有一次分配内存的调用，所以 100 个构造函数中只有第一次调用构造函数前把 onTheHeap 设置为 true。当调用第二个构造函数时，会抛出一个异常，你真倒霉。

即使不用数组，bit-setting 操作也会失败。考虑这条语句：

```
UPNumber *pn = new UPNumber(*new UPNumber);
```

这里我们在堆中建立两个 UPNumber，让 pn 指向其中一个对象；这个对象用另一个对象的值进行初始化。这个代码有一个内存泄漏，我们先忽略这个泄漏，这有利于下面对这条表达式的测试，执行它时会发生什么事情：

```
new UPNumber(*new UPNumber)
```

它包含 new 操作符的两次调用，因此要调用两次 operator new 和调用两次 UPNumber 构造函数（参见条款 8）。程序员一般期望这些函数以如下顺序执行：

调用第一个对象的 operator new
调用第一个对象的构造函数
调用第二个对象的 operator new
调用第二个对象的构造函数

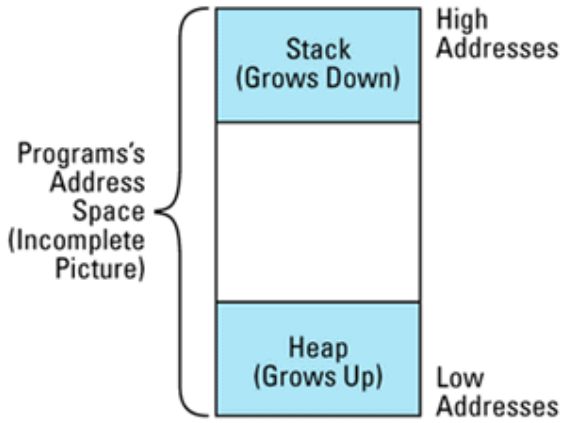
但是 C++语言没有保证这就是它调用的顺序。一些编译器以如下这种顺序生成函数调用：

调用第一个对象的 operator new
调用第二个对象的 operator new
调用第一个对象的构造函数
调用第二个对象的构造函数

编译器生成这种代码丝毫没有错，但是在 operator new 中 set-a-bit 的技巧无法与这种编译器一起使用。因为在第一步和第二步设置的 bit，第三步中被清除，那么在第四步调用对象的构造函数时，就会认为对象不再堆中，即使它确实在。

这些困难没有否定让每个构造函数检测*this 指针是否在堆中这个方法的核心思想，它们只是表明检测在 operator new(或 operator new[])里的 bit set 不是一个可靠的判断方法。我们需要更好的方法进行判断。

如果你陷入了极度绝望当中，你可能会沦落进不可移植的领域里。例如你决定利用一个在很多系统上存在的事实，程序的地址空间被做为线性地址管理，程序的栈从地址空间的顶部向下扩展，堆则从底部向上扩展：



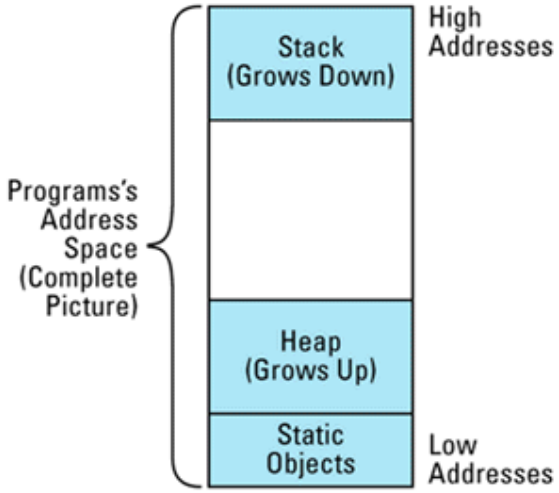
在以这种方法管理程序内存的系统里（很多系统都是，但是也有很多不是这样），你可能会想能够使用下面这个函数来判断某个特定的地址是否在堆中：

```
// 不正确的尝试，来判断一个地址是否在堆中
bool onHeap(const void *address)
{
  char onTheStack;                    // 局部栈变量
  return address < &onTheStack;
}
```

这个函数背后的思想很有趣。在 **onHeap** 函数中 **onTheSatck** 是一个局部变量。因此它在堆栈上。当调用 **onHeap** 时，它的栈框架（**stack frame**）**(**也就是它的 **activation record)** 被放在程序栈的顶端，因为栈在结构上是向下扩展的（趋向低地址），**onTheStack** 的地址肯定比任何栈中的变量或对象的地址小。如果参数 **address** 的地址小于 **onTheStack** 的地址，它就不会在栈上，而是肯定在堆上。

到目前为止，这种逻辑很正确，但是不够深入。最根本的问题是对象可以被分配在三个地方，而不是两个。是的，栈和堆能够容纳对象，但是我们忘了静态对象。静态对象是那些在程序运行时仅能初始化一次的对象。静态对象不仅仅包括显示地声明为 **static** 的对象，也包括在全局和命名空间里的对象（参见条款 **47**）。这些对象肯定位于某些地方，而这些地方既不是栈也不是堆。

它们的位置是依据系统而定的，但是在很多栈和堆相向扩展的系统里，它们位于堆的底端。先前内存管理的图片到讲述的是事实，而且是很多系统都具有的事实，但是没有告诉我们这些系统全部的事实，加上静态变量后，这幅图片如下所示：



onHeap 不能工作的原因立刻变得很清楚了，不能辨别堆对象与静态对象的区别：

```
void allocateSomeObjects()
{
  char *pc = new char;              // 堆对象: onHeap(pc)
                                    // 将返回 true
  char c;                           // 栈对象: onHeap(&c)
                                    // 将返回 false
  static char sc;                   // 静态对象: onHeap(&sc)
                                    // 将返回 true
  ...
}
```

现在你可能不顾一切地寻找区分堆对象与栈对象的方法，在走头无路时你想在可移植性上打主意，但是你会这么孤注一掷地进行一个不能获得正确结果的交易么？绝对不会。我知道你会拒绝使用这种虽然诱人但是不可靠的“地址比对”技巧。

令人伤心的是不仅没有一种可移植的方法来判断对象是否在堆上，而且连能在多数时间正常工作的“准可移植”的方法也没有。如果你实在非得必须判断一个地址是否在堆上，你必须使用完全不可移植的方法，其实现依赖于系统调用，只能这样做了。因此你最好重新设计你的软件，以便你可以不需要判断对象是否在堆中。

如果你发现自己实在为对象是否在堆中这个问题所困扰，一个可能的原因是你想知道对象是否能在其上安全调用 **delete**。这种删除经常采用“**delete this**”这种声明狼籍的形式。不过知道“是否能安全删除一个指针”与“只简单地知道一个指针是否指向堆中的事物”不一样，因为不是所有在堆中的事物都能被安全地 **delete**。再考虑包含 **UPNumber** 对象的 **Asset** 对象：

```
class Asset {
private:
  UPNumber value;
  ...
};
Asset *pa = new Asset;
```

很明显 ***pa**（包括它的成员 **value**）在堆上。同样很明显在指向 **pa->value** 上调用 **delete** 是不安全的，因为该指针不是被 **new** 返回的。

幸运的是“判断是否能够删除一个指针”比“判断一个指针指向的事物是否在堆上”要容易。因为对于前者我们只需要一个 **operator new** 返回的地址集合。因为我们能自己编写 **operator new** 函数（参见 **Effective C++**条款 **8**—条款 **10**），所以构建这样一个集合很容易。如下所示，我们这样解决这个问题：

```
void *operator new(size_t size)
{
  void *p = getMemory(size);          //调用一些函数来分配内存，
                                      //处理内存不够的情况
  把 p 加入到一个被分配地址的集合;
  return p;
}
void operator delete(void *ptr)
{
  releaseMemory(ptr);                 // return memory to
                                      // free store
  从被分配地址的集合中移去 ptr;
}
bool isSafeToDelete(const void *address)
{
  返回 address 是否在被分配地址的集合中;
}
```

这很简单，**operator new** 在地址分配集合里加入一个元素，**operator delete** 从集合中移去项目，**isSafeToDelete** 在集合中查找并确定某个地址是否在集合中。如果 **operator new** 和 **operator delete** 函数在全局作用域中，它就能适用于所有的类型，甚至是内建类型。

在实际当中，有三种因素制约着对这种设计方式的使用。第一是我们极不愿意在全局域定义任何东西，特别是那些已经具有某种含义的函数，象 **operator new** 和 **operator delete**。正如我们所知，只有一个全局域，只有一种具有正常特征形式（也就是参数类型）的 **operator new** 和 **operator delete**（条款 **9**）。这样做会使得我们的软件与其它也实现全局版本的 **operator new** 和 **operator delete** 的软件（例如许多面向对象数据库系统）不兼容。

我们考虑的第二个因素是效率：如果我们不需要这些，为什么还要为跟踪返回的地址而负担额外的开销呢？

最后一点可能有些平常，但是很重要。实现 **isSafeToDelete** 让它总能够正常工作是不可能的。难点是多继承下来的类或继承自虚基类的类有多个地址，所以无法保证传给 **isSafeToDelete** 的地址与 **operator new** 返回的地址相同，即使对象在堆中建立。有关细节参见条款 **M24** 和条款 **M31**。

我们希望这些函数提供这些功能时能够不污染全局命名空间，没有额外的开销，没有正确性问题。幸运的是 **C++**使用一种抽象 **mixin** 基类满足了我们的需要。

抽象基类是不能被实例化的基类，也就是至少具有一个纯虚函数的基类。**mixin**（“**mix in**”）类提供某一特定的功能，并可以与其继承类提供的其它功能相兼容（参见 **Effective C++**条款 **7**）。这种类几乎都是抽象类。因此我们能够使用抽象混合（**mixin**）基类给派生类提供判断指针指向的内存是否由 **operator new** 分配的能力。该类如下所示：

```
class HeapTracked {                   // 混合类; 跟踪
public:                               // 从 operator new 返回的 ptr
  class MissingAddress{};             // 异常类，见下面代码
  virtual ~HeapTracked() = 0;
  static void *operator new(size_t size);
  static void operator delete(void *ptr);
  bool isOnHeap() const;
private:
  typedef const void* RawAddress;
  static list<RawAddress> addresses;
};
```

这个类使用了 **list**（链表）数据结构跟踪从 **operator new** 返回的所有指针，**list** 是标准 **C++**库的一部分（参见 **Effective C++**条款 **49** 和本书条款 **35**）。**operator new** 函数分配内存并把地址加入到 **list** 中；**operator delete** 用来释放内存并从 **list** 中移去地址元素。

isOnHeap 判断一个对象的地址是否在 list 中。

HeapTracked 类的实现很简单，调用全局的 operator new 和 operator delete 函数来完成内存的分配与释放，list 类里的函数进行插入操作和删除操作，并进行单语句的查找操作。以下是 HeapTracked 的全部实现：

```
// mandatory definition of static class member
list<RawAddress> HeapTracked::addresses;
// HeapTracked 的析构函数是纯虚函数，使得该类变为抽象类。
// (参见 Effective C++条款 14). 然而析构函数必须被定义，
//所以我们做了一个空定义。.
HeapTracked::~HeapTracked() {}
void * HeapTracked::operator new(size_t size)
{
  void *memPtr = ::operator new(size);  // 获得内存
  addresses.push_front(memPtr);         // 把地址放到 list 的前端
  return memPtr;
}
void HeapTracked::operator delete(void *ptr)
{
  //得到一个 "iterator"，用来识别 list 元素包含的 ptr；
  //有关细节参见条款 35
  list<RawAddress>::iterator it =
    find(addresses.begin(), addresses.end(), ptr);
  if (it != addresses.end()) {       // 如果发现一个元素
    addresses.erase(it);             //则删除该元素
    ::operator delete(ptr);          // 释放内存
  } else {                           // 否则
    throw MissingAddress();          // ptr 就不是用 operator new
  }                                  // 分配的，所以抛出一个异常
}
bool HeapTracked::isOnHeap() const
{
  // 得到一个指针，指向*this 占据的内存空间的起始处，
  // 有关细节参见下面的讨论
  const void *rawAddress = dynamic_cast<const void*>(this);
  // 在 operator new 返回的地址 list 中查到指针
  list<RawAddress>::iterator it =
    find(addresses.begin(), addresses.end(), rawAddress);
  return it != addresses.end();      // 返回 it 是否被找到
}
```

尽管你可能对 list 类和标准 C++库的其它部分不很熟悉，代码还是很一目了然。条款 M35 将解释这里的每件东西，不过代码里的注释已经能够解释这个例子是如何运行的。

只有一个地方可能让你感到困惑，就是这个语句（在 isOnHeap 函数中）

```
const void *rawAddress = dynamic_cast<const void*>(this);
```

我前面说过带有多继承或虚基类的对象会有几个地址，这导致编写全局函数 isSafeToDelete 会很复杂。这个问题在 isOnHeap 中仍然会遇到，但是因为 isOnHeap 仅仅用于 HeapTracked 对象中，我们能使用 dynamic_cast 操作符（参见条款 M2）的一种特殊的特性来消除这个问题。只需简单地放入 dynamic_cast，把一个指针 dynamic_cast 成 void* 类型（或 const void*或 volatile void* 。。。。。），生成的指针将指向“原指针指向对象内存”的开始处。但是 dynamic_cast 只能用于“指向至少具有一个虚拟函数的对象”的指针上。我们该死的 isSafeToDelete 函数可以用于指向任何类型的指针，所以 dynamic_cast 也不能帮助它。isOnHeap 更具有选择性（它只能测试指向 HeapTracked 对象的指针），所以

能把 **this** 指针 **dynamic_cast** 成 **const void***，变成一个指向当前对象起始地址的指针。如果 **HeapTracked::operator new** 为当前对象分配内存，这个指针就是 **HeapTracked::operator new** 返回的指针。如果你的编译器支持 **dynamic_cast** 操作符，这个技巧是完全可移植的。

使用这个类，即使是最初级的程序员也可以在类中加入跟踪堆中指针的功能。他们所需要做的就是让他们的类从 **HeapTracked** 继承下来。例如我们想判断 **Assert** 对象指针指向的是否是堆对象：

```
class Asset: public HeapTracked {
private:
  UPNumber value;
  ...
};
```

我们能够这样查询 **Assert***指针，如下所示：

```
void inventoryAsset(const Asset *ap)
{
  if (ap->isOnHeap()) {
    ap is a heap-based asset — inventory it as such;
  }
  else {
    ap is a non-heap-based asset — record it that way;
  }
}
```

象 **HeapTracked** 这样的混合类有一个缺点，它不能用于内建类型，因为象 **int** 和 **char** 这样的类型不能继承自其它类型。不过使用象 **HeapTracked** 的原因一般都是要判断是否可以调用“**delete this**”，你不可能在内建类型上调用它，因为内建类型没有 **this** 指针。

! 禁止堆对象

判断对象是否在堆中的测试到现在就结束了。与此相反的领域是“禁止在堆中建立对象”。通常对象的建立这样三种情况：对象被直接实例化；对象做为派生类的基类被实例化；对象被嵌入到其它对象内。我们将按顺序地讨论它们。

禁止用户直接实例化对象很简单，因为总是调用 **new** 来建立这种对象，你能够禁止用户调用 **new**。你不能影响 **new** 操作符的可用性（这是内嵌于语言的），但是你能够利用 **new** 操作符总是调用 **operator new** 函数这点（参见条款 **M8**），来达到目的。你可以自己声明这个函数，而且你可以把它声明为 **private**。例如，如果你想不想让用户在堆中建立 **UPNumber** 对象，你可以这样编写：

```
class UPNumber {
private:
  static void *operator new(size_t size);
  static void operator delete(void *ptr);
  ...
};
```

现在用户仅仅可以做允许它们做的事情：

```
UPNumber n1;                          // okay
static UPNumber n2;                   // also okay
UPNumber *p = new UPNumber;           // error! attempt to call
                                      // private operator new
```

把 **operator new** 声明为 **private** 就足够了，但是把 **operator new** 声明为 **private**，而把 **iperator delete** 声明为 **public**，这样做有些怪异，所以除非有绝对需要的原因，否则不要把它们分开声明，最好在类的一个部分里声明它们。如果你也想禁止 **UPNumber** 堆对象数组，可以把 **operator new[]**和 **operator delete[]**（参见条款 **M8**）也声明为 **private**。（**operator new** 和 **operator delete** 之间的联系比大多数人所想象的要强得多。有关它们之间关系的鲜为人知的一面，可以参见我的文章 **counting objects** 里的 **sidebar** 部分。）

有趣的是，把 operator new 声明为 private 经常会阻碍 UPNumber 对象做为一个位于堆中的派生类对象的基类被实例化。因为 operator new 和 operator delete 是自动继承的，如果 operator new 和 operator delete 没有在派生类中被声明为 public（进行改写，overwrite），它们就会继承基类中 private 的版本，如下所示：

```
class UPNumber { ... };              // 同上
class NonNegativeUPNumber:           //假设这个类
  public UPNumber {                  //没有声明 operator new
  ...
};
NonNegativeUPNumber n1;              // 正确
static NonNegativeUPNumber n2;       // 也正确
NonNegativeUPNumber *p =             // 错误! 试图调用
  new NonNegativeUPNumber;           // private operator new
```

如果派生类声明它自己的 operator new，当在堆中分配派生对象时，就会调用这个函数，于是得另找一种不同的方法来防止 UPNumber 基类的分配问题。UPNumber 的 operator new 是 private 这一点，不会对包含 UPNumber 成员对象的对象的分配产生任何影响：

```
class Asset {
public:
  Asset(int initValue);
  ...
private:
  UPNumber value;
};
Asset *pa = new Asset(100);          // 正确，调用
                                     // Asset::operator new 或
                                     // ::operator new，不是
                                     // UPNumber::operator new
```

实际上，我们又回到了这个问题上来，即“如果 UPNumber 对象没有被构造在堆中，我们想抛出一个异常”。当然这次的问题是“如果对象在堆中，我们想抛出异常”。正像没有可移植的方法来判断地址是否在堆中一样，也没有可移植的方法判断地址是否不在堆中，所以我们很不走运，不过这也丝毫不奇怪，毕竟如果我们能辨别出某个地址在堆上，我们也就能辨别出某个地址不在堆上。但是我们什么都不能辨别出来。

7.4 Item M28：灵巧（smart）指针

灵巧指针是一种外观和行为都被设计成与内建指针相类似的对象，不过它能提供更多的功能。它们有许多应用的领域，包括资源管理（参见条款 M9、M10、M25 和 M31）和重复代码任务的自动化（参见条款 M17 和 M29）。

当你使用灵巧指针替代 C++的内建指针（也就是 dumb pointer），你就能控制下面这些方面的指针的行为：

- 构造和析构。你可以决定建立灵巧指针时应该怎么做。通常赋给灵巧指针缺省值 0，避免出现令人头疼的未初始化的指针。当指向某一对象的最后一个灵巧指针被释放时，某些灵巧指针被设计成负责删除它们指向的对象。这样做对防止资源泄漏很有帮助。
- 拷贝和赋值。你能对拷贝灵巧指针或有灵巧指针参与的赋值操作进行控制。对于某些类型的灵巧指针来说，期望的行为是自动拷贝它们所指向的对象或用对这些对象进行赋值操作，也就是进行 deep copy(深层拷贝)。对于其它的一些灵巧指针来说，仅仅拷贝指针本身或对指针进行赋值操作。还有一部分类型的灵巧指针根本就不允许这些操作。无论你认为应该如何去做，灵巧指针始终受你的控制。
- Dereferencing(取出指针所指东西的内容)。当用户引用被灵巧指针所指的对象，会发生什么事情呢？你可以自行决定。例如你可以用灵巧指针实现条款 M17 提到的

lazy fetching 方法。

灵巧指针从模板中生成，因为要与内建指针类似，必须是 strongly typed(强类型)的；模板参数确定指向对象的类型。大多数灵巧指针模板看起来都象这样：

```
template<class T>
//灵巧指针对象模板
class SmartPtr {
public:
  SmartPtr(T* realPtr = 0);          // 建立一个灵巧指针
                                     // 指向 dumb pointer 所指的
                                     // 对象。未初始化的指针
                                     // 缺省值为 0(null)
  SmartPtr(const SmartPtr& rhs);     // 拷贝一个灵巧指针
  ~SmartPtr();                       // 释放灵巧指针
  // make an assignment to a smart ptr
  SmartPtr& operator=(const SmartPtr& rhs);
  T* operator->() const;             // dereference 一个灵巧指针
                                     // 以访问所指对象的成员
  T& operator*() const;              // dereference 灵巧指针
private:
  T *pointee;                        // 灵巧指针所指的对象
};
```

拷贝构造函数和赋值操作符都被展现在这里。对于灵巧指针类来说，不能允许进行拷贝和赋值操作，它们应该被声明为 private（参见 Effective C++条款 27）。两个 dereference 操作符被声明为 const，是因为 dereference 一个灵巧指针时不会对其自身进行修改（尽管可以修改指针所指的对象）（这样，const 型的灵巧指针才可以使用这两个操作符。）。最后，每个指向 T 对象的灵巧指针包含一个指向 T 的 dumb pointer。这个 dumb pointer 指向的对象才是灵巧指针指向的真正对象。

进入灵巧指针实现的细节之前，应该研究一下用户如何使用灵巧指针。考虑一下，存在一个分布式系统（即其上的对象一些在本地，一些在远程）。相对于访问远程对象，访问本地对象通常总是又简单而又速度快，因为远程访问需要远程过程调用（RPC），或其它一些联系远距离计算机的方法。对于编写程序代码的用户来说，采用不同的方法分别处理本地对象与远程对象是一件很烦人的事情。让所有的对象看起来都位于一个地方会更方便。灵巧指针可以让程序库实现这样的梦想。

```
template<class T>                    // 指向位于分布式 DB（数据库）
class DBPtr {                        // 中对象的灵巧指针模板
public:                              //
  DBPtr(T *realPtr = 0);             // 建立灵巧指针，指向
                                     // 由一个本地 dumb pointer
                                     // 给出的 DB 对象
  DBPtr(DataBaseID id);              // 建立灵巧指针，
                                     // 指向一个 DB 对象，
                                     // 具有惟一的 DB 识别符
  ...                                // 其它灵巧指针函数
};                                   //同上
class Tuple {                        // 数据库元组类
public:
  ...
  void displayEditDialog();          // 显示一个图形对话框，
                                     // 允许用户编辑元组。
                                     // user to edit the tuple
  bool isValid() const;              // 返回*this 是否通过了
};                                   // 合法性验证
```

```
// 这个类模板用于在修改 T 对象时进行日志登记。
// 有关细节参见下面的叙述：
template<class T>
class LogEntry {
public:
  LogEntry(const T& objectToBeModified);
  ~LogEntry();
};
void editTuple(DBPtr<Tuple>& pt)
{
  LogEntry<Tuple> entry(*pt);        // 为这个编辑操作登记日志
                                     // 有关细节参见下面的叙述
  // 重复显示编辑对话框，直到提供了合法的数值。
  do {
    pt->displayEditDialog();
  } while (pt->isValid() == false);
}
```

在 **editTuple** 中被编辑的元组物理上可以位于本地也可以位于远程，但是编写 **editTuple** 的程序员不用关心这些事情。灵巧指针类隐藏了系统的这些方面。就程序员所关心的方面而言，通过灵巧指针对象进行访问元组，除了如何声明它们不同外，其行为就像一个内建指针。

注意在 **editTuple** 中 **LogEntry** 对象的用法。一种更传统的设计是在调用 **displayEditDialog** 前开始日志记录，调用后结束日志记录。在这里使用的方法是让 **LogEntry** 的构造函数启动日志记录，析构函数结束日志记录。正如条款 **M9** 所解释的，当面对异常时，让对象自己开始和结束日志记录比显示地调用函数可以使得程序更健壮。而且建立一个 **LogEntry** 对象比每次都调用开始记录和结束记录函数更容易。

正如你所看到的，使用灵巧指针与使用 **dump pointer** 没有很大的差别。这表明了封装是非常有效的。灵巧指针的用户可以象使用 **dumb pointer** 一样使用灵巧指针。正如我们将看到的，有时这种替代会更透明化。

I　灵巧指针的构造、赋值和析构

灵巧指针的的构造通常很简单：找到指向的对象（一般由灵巧指针构造函数的参数给出），让灵巧指针的内部成员 **dumb pointer** 指向它。如果没有找到对象，把内部指针设为 **0** 或发出一个错误信号（可以是抛出一个异常）。

灵巧指针拷贝构造函数、赋值操作符函数和析构函数的实现由于（所指对象的）所有权的问题所以有些复杂。如果一个灵巧指针拥有它指向的对象，当它被释放时必须负责删除这个对象。这里假设灵巧指针指向的对象是动态分配的。这种假设在灵巧指针中是常见的（有关确定这种假设是真实的方法，参见条款 **M27**）。

看一下标准 **C++**类库中 **auto_ptr** 模板。这如条款 **M9** 所解释的，一个 **auto_ptr** 对象是一个指向堆对象的灵巧指针，直到 **auto_ptr** 被释放。**auto_ptr** 的析构函数删除其指向的对象时，会发生什么事情呢？**auto_ptr** 模板的实现如下：

```
template<class T>
class auto_ptr {
public:
  auto_ptr(T *ptr = 0): pointee(ptr) {}
  ~auto_ptr() { delete pointee; }
  ...
private:
  T *pointee;
};
```

假如 **auto_ptr** 拥有对象时，它可以正常运行。但是当 **auto_ptr** 被拷贝或被赋值时，会发生什么情况呢？

```
auto_ptr<TreeNode> ptn1(new TreeNode);
auto_ptr<TreeNode> ptn2 = ptn1;      // 调用拷贝构造函数
                                     //会发生什么情况？
auto_ptr<TreeNode> ptn3;
ptn3 = ptn2;                         // 调用 operator=;
                                     // 会发生什么情况?
```

如果我们只拷贝内部的 **dumb pointer**，会导致两个 **auto_ptr** 指向一个相同的对象。这是一个灾难，因为当释放 **quto_ptr** 时每个 **auto_ptr** 都会删除它们所指的对象。这意味着一个对象会被我们删除两次。这种两次删除的结果将是不可预测的（通常是灾难性的）。

另一种方法是通过调用 **new**，建立一个所指对象的新拷贝。这确保了不会有许多指向同一个对象的 **auto_ptr**，但是建立（以后还得释放）新对象会造成不可接受的性能损耗。并且我们不知道要建立什么类型的对象，因为 **auto_ptr<T>**对象不用必须指向类型为 **T** 的对象，它也可以指向 **T** 的派生类型对象。虚拟构造函数（参见条款 **M25**）可能帮助我们解决这个问题，但是好象不能把它们用在 **auto_ptr** 这样的通用类中。

如果 **quto_ptr** 禁止拷贝和赋值，就可以消除这个问题，但是采用“当 **auto_ptr** 被拷贝和赋值时，对象所有权随之被传递”的方法，是一个更具灵活性的解决方案：

```
template<class T>
class auto_ptr {
public:
  ...
  auto_ptr(auto_ptr<T>& rhs);        // 拷贝构造函数
  auto_ptr<T>&                       // 赋值
  operator=(auto_ptr<T>& rhs);       // 操作符
  ...
};
template<class T>
auto_ptr<T>::auto_ptr(auto_ptr<T>& rhs)
{
  pointee = rhs.pointee;             // 把*pointee 的所有权
                                     // 传递到 *this
  rhs.pointee = 0;                   // rhs 不再拥有
}                                    // 任何东西
template<class T>
auto_ptr<T>& auto_ptr<T>::operator=(auto_ptr<T>& rhs)
{
  if (this == &rhs)                  // 如果这个对象自我赋值
    return *this;                    // 什么也不要做
  delete pointee;                    // 删除现在拥有的对象
  pointee = rhs.pointee;             // 把*pointee 的所有权
  rhs.pointee = 0;                   // 从 rhs 传递到 *this
  return *this;
}
```

注意赋值操作符在接受新对象的所有权以前必须删除原来拥有的对象。如果不这样做，原来拥有的对象将永远不会被删除。记住，除了 **auto_ptr** 对象，没有人拥有 **auto_ptr** 指向的对象。

因为当调用 **auto_ptr** 的拷贝构造函数时，对象的所有权被传递出去，所以通过传值方式传递 **auto_ptr** 对象是一个很糟糕的方法。因为：

```
// 这个函数通常会导致灾难发生
void printTreeNode(ostream& s, auto_ptr<TreeNode> p)
{ s << *p; }
int main()
```

```
{
  auto_ptr<TreeNode> ptn(new TreeNode);
  ...
  printTreeNode(cout, ptn);          //通过传值方式传递 auto_ptr
  ...
}
```

当 printTreeNode 的参数 p 被初始化时（调用 auto_ptr 的拷贝构造函数），ptn 指向对象的所有权被传递到给了 p。当 printTreeNode 结束执行后，p 离开了作用域，它的析构函数删除它指向的对象（就是原来 ptr 指向的对象）。然而 ptr 已不再指向任何对象（它的 dumb pointer 是 null），所以调用 printTreeNode 以后任何试图使用它的操作都将产生未定义的行为。只有在你确实想把对象的所有权传递给一个临时的函数参数时，才能通过传值方式传递 auto_ptr。这种情况很少见。

这不是说你不能把 auto_ptr 做为参数传递，这只意味着不能使用传值的方法。通过 const 引用传递（Pass-by-reference-to-const）的方法是这样的：

```
// 这个函数的行为更直观一些
void printTreeNode(ostream& s,
                   const auto_ptr<TreeNode>& p)
{ s << *p; }
```

在函数里，p 是一个引用，而不是一个对象，所以不会调用拷贝构造函数初始化 p。当 ptn 被传递到上面这个 printTreeNode 时，它还保留着所指对象的所有权，调用 printTreeNode 以后还可以安全地使用 ptn。所以通过 const 引用传递 auto_ptr 可以避免传值所产生的风险。（“引用传递”替代“传值”的其他原因参见 Effective C++条款 22）。

在拷贝和赋值中，把对象的所有权从一个灵巧指针传递到另一个中去，这种思想很有趣，而且你可能已经注意到拷贝构造函数和赋值操作符不同寻常的声明方法同样也很有趣。这些函数的参数通常会带有 const，但是上面这些函数则没有。实际上在拷贝和赋值中上述这些函数的代码修改了这些参数。也就是说，如果 auto_ptr 对象被拷贝或做为赋值操作的数据源，就会修改 auto_ptr 对象！

是的，就是这样。C++是如此灵活能让你这样去做，真是太好了。如果语言要求拷贝构造函数和赋值操作符必须带有 const 参数，你必须去掉参数的 const 属性（参见 Effective C++条款 21）或用其他方法实现所有权的转移。准确地说：当拷贝一个对象或这个对象做为赋值的数据源，就会修改该对象。这可能有些不直观，但是它是简单的、直接的，在这种情况下也是准确的。

如果你发现研究这些 auto_ptr 成员函数很有趣，你可能希望看看完整的实现。在 291 页至 294 页上有（指原书页码），在那里你也能看到，在标准 C++库中 auto_ptr 模板有比这里所描述的更灵活的拷贝构造函数和赋值操作符。在标准 C++库中，这些函数是成员函数模板，而不只是成员函数。（在本条款的后面会讲述成员函数模板。也可以阅读 Effective C++条款 25）。

灵巧指针的析构函数通常是这样的：

```
template<class T>
SmartPtr<T>::~SmartPtr()
{
  if (*this owns *pointee) {
    delete pointee;
  }
}
```

有时删除前不需要进行测试，例如在一个 auto_ptr 总是拥有它指向的对象时。而在另一些时候，测试会更为复杂：一个使用了引用计数（参见条款 M29）灵巧指针必须在判断是否有权删除所指对象前调整引用计数值。当然还有一些灵巧指针象 dumb pointer 一样，当它们被删除时，对所指对象没有任何影响。

实现 Dereference 操作符

让我们把注意力转向灵巧指针的核心部分，operator*和 operator->函数。前者返回所

指的对象。理论上，这很简单：

```
template<class T>
    T& SmartPtr<T>::operator*() const
    {
      perform "smart pointer" processing;
      return *pointee;
    }
```

首先，无论函数做什么，必须先初始化指针或使 **pointee** 合法。例如，如果使用 **lazy fetch**（参见条款 **M17**），函数必须为 **pointee** 建立一个新对象。一旦 **pointee** 合法了，**operator*** 函数就返回其所指对象的一个引用。

注意返回类型是一个引用。如果返回对象，尽管编译器允许这么做，却可能导致灾难性后果。必须时刻牢记：**pointee** 不用必须指向 **T** 类型对象；它也可以指向 **T** 的派生类对象。如果在这种情况下 **operator*** 函数返回的是 **T** 类型对象而不是派生类对象的引用，你的函数实际上返回的是一个错误类型的对象！（这是一个 **slicing**（切割）问题，参见 **Effective C++** 条款 **22** 和本书条款 **13**。）在返回的这种对象上调用虚拟函数，不会触发与（原先）所指对象的动态类型相符的函数。实际上就是说你的灵巧指针将不能支持虚拟函数，象这样的指针再灵巧也没有用。而返回一个引用还能够具有更高的效率（不需要构造一个临时对象，参见条款 **M19**）。能够兼顾正确性与效率当然是一件好事。

如果你是一个急性子的人，你可能会想如果一些人在 **null** 灵巧指针上调用 **operator***，也就是说灵巧指针的 **dumb pointer** 是 **null**。放松。随便做什么都行。**dereference** 一个空指针的结果是未定义的，所以随你怎么实现都不算错。想抛一个异常么？可以，抛出吧。想调用 **abort** 函数（可能被 **assert** 在失败时调用）？好的，调用吧。想遍历内存把每个字节都设成你生日与 **256** 模数么？当然也可以。虽说这样做没有什么好处，但是就语言本身而言，你完全是自由的。

operator->的情况与 **operator***是相同的，但是在分析 **operator->**之前，让我们先回忆一下这个函数调用的与众不同的含义。再考虑 **editTuple** 函数，其使用一个指向 **Tuple** 对象的灵巧指针：

```
    void editTuple(DBPtr<Tuple>& pt)
    {
      LogEntry<Tuple> entry(*pt);
      do {
        pt->displayEditDialog();
      } while (pt->isValid() == false);
    }
```

语句

```
    pt->displayEditDialog();
```

被编译器解释为：

```
    (pt.operator->())->displayEditDialog();
```

这意味着不论 **operator->**返回什么，它必须在返回结果上使用 **member-selection operator(**成员选择操作符**)**（**->**）。因此 **operator->**仅能返回两种东西：一个指向某对象的 **dumb pointer** 或另一个灵巧指针。多数情况下，你想返回一个普通 **dumb pointer**。在此情况下，你这样实现 **operator->** **:**

```
    template<class T>
    T* SmartPtr<T>::operator->() const
    {
      perform "smart pointer" processing;
      return pointee;
    }
```

这样做运行良好。因为该函数返回一个指针，通过 **operator->**调用虚拟函数，其行为也是正确的。

对于很多程序来说，这就是你需要了解灵巧指针的全部东西。条款 **M29** 的引用计数代码

并没有比这里更多的功能。但是如果你想更深入地了解灵巧指针，你必须知道更多的有关 **dumb pointer** 的知识和灵巧指针如何能或不能进行模拟 **dumb pointer**。如果你的座右铭是 **“Most people stop at the Z-but not me(**多数人浅尝而止，但我不能够这样**)** ”，下面讲述的内容正适合你

I 测试灵巧指针是否为 NULL

目前为止我们讨论的函数能让我们建立、释放、拷贝、赋值、**dereference** 灵巧指针。但是有一件我们做不到的事情是“发现灵巧指针为 **NULL**”：

```
SmartPtr<TreeNode> ptn;
...
if (ptn == 0) ...                    // error!
if (ptn) ...                         // error!
if (!ptn) ...                        // error!
```

这是一个严重的限制。

在灵巧指针类里加入一个 **isNull** 成员函数是一件很容易的事，但是没有解决当测试 **NULL** 时灵巧指针的行为与 **dumb pointer** 不相似的问题。另一种方法是提供隐式类型转换操作符，允许编译上述的测试。一般应用于这种目的的类型转换是 **void*** ：

```
template<class T>
class SmartPtr {
public:
  ...
  operator void*();                  // 如果灵巧指针为 null，
  ...                                // 返回 0， 否则返回
};                                   // 非 0。
SmartPtr<TreeNode> ptn;
...
if (ptn == 0) ...                    // 现在正确
if (ptn) ...                         // 也正确
if (!ptn) ...                        // 正确
```

这与 **iostream** 类中提供的类型转换相同，所以可以这样编写代码：

```
ifstream inputFile("datafile.dat");
if (inputFile) ...                   // 测试 inputFile 是否已经被
                                     // 成功地打开。
```

象所有的类型转换函数一样，它有一个缺点：在一些情况下虽然大多数程序员希望它调用失败，但是函数确实能够成功地被调用（参见条款 **M5**）。特别是它允许灵巧指针与完全不同的类型之间进行比较：

```
SmartPtr<Apple> pa;
SmartPtr<Orange> po;
...
if (pa == po) ...                    // 这能够被成功编译!
```

即使在 **SmartPtr<Apple>** 和 **SmartPtr<Orange>**之间没有 **operator=** 函数，也能够编译，因为灵巧指针被隐式地转换为 **void***指针，而对于内建指针类型有内建的比较函数。这种进行隐式类型转换的行为特性很危险。（再回看一下条款 **M5,**必须反反复复地阅读，做到耳熟能详。）

在 **void***类型转换方面，也有一些变化。有些设计者采用到 **const void***的类型转换，还有一些采取转换到 **bool** 的方法。这些变化都没有消除混合类型比较的问题。

有一种两全之策可以提供合理的测试 **null** 值的语法形式，同时把不同类型的灵巧指针之间进行比较的可能性降到最低。这就是在灵巧指针类中重载 **operator!**，当且仅当灵巧指针是一个空指针时，**operator!**返回 **true**：

```
template<class T>
class SmartPtr {
public:
```

```
  ...
  bool operator!() const;            // 当且仅当灵巧指针是
  ...                                // 空值，返回 true。
};
```

用户程序如下所示：

```
SmartPtr<TreeNode> ptn;
...
if (!ptn) {                          // 正确
  ...                                // ptn 是空值
}
else {
  ...                                // ptn 不是空值
}
```

但是这样就不正确了：

```
if (ptn == 0) ...                    // 仍然错误
if (ptn) ...                         // 也是错误的
```

仅在这种情况下会存在不同类型之间进行比较：

```
SmartPtr<Apple> pa;
SmartPtr<Orange> po;
...
if (!pa == !po) ...                 // 能够编译
```

幸好程序员不会经常这样编写代码。有趣的是，**iostream** 库的实现除了提供 **void***隐式的类型转换，也有 **operator!**函数，不过这两个函数通常测试的流状态有些不同。（在 **C++** 类库标准中（参见 **Effective C++** 条款 **49** 和本书条款 **M35**），**void***隐式的类型转换已经被 **bool** 类型的转换所替代，**operator bool** 总是返回与 **operator!**相反的值。）

I　把灵巧指针转变成 dumb 指针

有时你要在一个程序里或已经使用 **dumb** 指针的程序库中添加灵巧指针。例如，你的分布式数据库系统原来不是分布式的，所以可能有一些老式的库函数没有使用灵巧指针：

```
class Tuple { ... };                 // 同上
void normalize(Tuple *pt);           // 把*pt 放入
                                     // 范式中；注意使用的
                                     // 是 dumb 指针
```

考虑一下，如果你试图用指向 **Tuple** 的灵巧指针作参数调用 **normalize**，会出现什么情况：

```
DBPtr<Tuple> pt;
...
normalize(pt);                       // 错误!
```

这种调用不能够编译，因为不能把 **DBPtr<Tuple>**转换成 **Tuple***。你可以这样做，从而使该该函数正常运行：

```
normalize(&*pt);                     // 繁琐，但合法
```

不过我觉得你会讨厌这种调用方式。

在灵巧指针模板中增加指向 **T** 的 **dumb** 指针的隐式类型转换操作符，可以让以上函数调用成功运行：

```
template<class T>                   // 同上
class DBPtr {
public:
  ...
  operator T*() { return pointee; }
  ...
};
DBPtr<Tuple> pt;
```

```
...
normalize(pt);                          // 能够运行
```

并且这个函数也消除了测试空值的问题：

```
if (pt == 0) ...                        // 正确，把 pt 转变成
                                        // Tuple*
if (pt) ...                             // 同上
if (!pt) ...                            // 同上 (reprise)
```

然而，它也有类型转换函数所具有的缺点（几乎总是这样，看条款 M5）。它使得用户能够很容易地直接访问 dumb 指针，绕过了“类指针（pointer-like）”对象所提供的“灵巧”特性：

```
void processTuple(DBPtr<Tuple>& pt)
{
  Tuple *rawTuplePtr = pt;              // 把 DBPtr<Tuple> 转变成
                                        // Tuple*
  使用 raw TuplePtr 修改 tuple;
}
```

通常，灵巧指针提供的“灵巧”行为特性是设计中的主要组成部分，所以允许用户使用 dumb 指针会导致灾难性的后果。例如，如果 DBPtr 实现了条款 M29 中引用计数的功能，允许客户端直接对 dumb 指针进行操作很可能破坏“引用计数”数据结构，而导致引用计数错误。

甚至即使你提供一个从灵巧指针到 dumb 指针的隐式转换操作符，灵巧指针也不能真正地做到与 dumb 指针互换。因为从灵巧指针到 dumb 指针的转换是“用户定义类型转换”，在同一时间编译器进行这种转换的次数不能超过一次。例如假设有一个表示所有能够访问某一元组的用户的类：

```
class TupleAccessors {
public:
  TupleAccessors(const Tuple *pt);    // pt identifies the
  ...                                 // tuple whose accessors
};                                    // we care about
```

通常，TupleAccessors 的单参数构造函数也可以作为从 Tuple*到 TupleAccessors 的类型转换操作符（参见条款 M5）。现在考虑一下用于合并两个 TupleAccessors 对象内信息的函数：

```
TupleAccessors merge(const TupleAccessors& ta1,
                     const TupleAccessors& ta2);
```

因为一个 Tuple*可以被隐式地转换为 TupleAccessors，用两个 dumb Tuple*调用 merge 函数，可以正常运行：

```
Tuple *pt1, *pt2;
...
merge(pt1, pt2);                        // 正确，两个指针被转换为
                                        // TupleAccessors objects
```

如果用灵巧指针 DBPtr<Tuple>进行调用，编译就会失败：

```
DBPtr<Tuple> pt1, pt2;
...
merge(pt1, pt2);                  // 错误!不能把 pt1 和
                                  // pt2 转换称 TupleAccessors 对象
```

因为从 DBPtr<Tuple>到 TupleAccessors 的转换要调用两次用户定义类型转换（一次从 DBPtr<Tuple>到 Tuple*，一次从 Tuple*到 TupleAccessors），编译器不会进行这种序列的转换。

提供到 dumb 指针的隐式类型转换的灵巧指针类也暴露了一个非常有害的 bug。考虑这个代码：

```
DBPtr<Tuple> pt = new Tuple;
```

```
...
delete pt;
```

这段代码应该不能被编译，**pt** 不是指针，它是一个对象，你不能删除一个对象。只有指针才能被删除，对么？

当然对了。但是回想一下条款 **M5**：编译器使用隐式类型转换来尽可能使函数调用成功；再回想一下条款 **M8**：使用 **delete** 会调用析构函数和 **operator delete**，两者都是函数。编译器为了使在 **delete pt** 语句里的两个函数成功调用，就把 **pt** 隐式转换为 **Tuple***，然后删除它。（本来是你写错了代码，而现在却编译过了，）这样做必然会破坏你的程序。

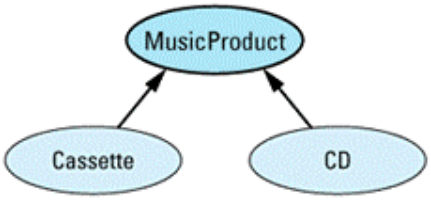
如果 **pt** 拥有它指向的对象，对象就会被删除两次，一次在调用 **delete** 时，第二次在 **pt** 的析构函数被调用时。当 **pt** 不拥有对象，而是其它人拥有时，如果拥有者同时负责删除 **pt** 的则情况还好，但是如果拥有者不是负责删除 **pt** 的人，可以预料它以后还会再次删除该对象。不论是最前者所述的情况还是最后者的情况都会导致一个对象被删除两次，这样做会产生不能预料的后果。

这个 **bug** 极为有害，因为隐藏在灵巧指针后面的全部思想就是让它们不论是在外观上还是在使用感觉上都与 **dumb** 指针尽可能地相似。你越接近这种思想，你的用户就越可能忘记正在使用灵巧指针。如果他们忘记了正在使用灵巧指针，肯定会在调用 **new** 之后调用 **delete**，以防止资源泄漏，谁又能责备他们这样做不对呢？

底线很简单：除非有一个让人非常信服的原因去这样做，否则绝对不要提供转换到 **dumb** 指针的隐式类型转换操作符。

l 灵巧指针和继承类到基类的类型转换

假设我们有一个 **public** 继承层次结构，以模型化音乐商店的商品：




```
class MusicProduct {
public:
  MusicProduct(const string& title);
  virtual void play() const = 0;
  virtual void displayTitle() const = 0;
  ...
};
class Cassette: public MusicProduct {
public:
  Cassette(const string& title);
  virtual void play() const;
  virtual void displayTitle() const;
  ...
};
class CD: public MusicProduct {
public:
  CD(const string& title);
  virtual void play() const;
  virtual void displayTitle() const;
  ...
};
```

再接着假设，我们有一个函数，给它一个 **MusicProduct** 对象，它能显示产品名，并播

放它：

```
void displayAndPlay(const MusicProduct* pmp, int numTimes)
{
  for (int i = 1; i <= numTimes; ++i) {
    pmp->displayTitle();
    pmp->play();
  }
}
```

这个函数能够这样使用：

```
Cassette *funMusic = new Cassette("Alapalooza");
CD *nightmareMusic = new CD("Disco Hits of the 70s");
displayAndPlay(funMusic, 10);
displayAndPlay(nightmareMusic, 0);
```

这并没有什么值得惊讶的东西，但是当我们用灵巧指针替代 **dumb** 指针，会发生什么呢：

```
void displayAndPlay(const SmartPtr<MusicProduct>& pmp,
                    int numTimes);
SmartPtr<Cassette> funMusic(new Cassette("Alapalooza"));
SmartPtr<CD> nightmareMusic(new CD("Disco Hits of the 70s"));
displayAndPlay(funMusic, 10);         // 错误!
displayAndPlay(nightmareMusic, 0);    // 错误!
```

既然灵巧指针这么聪明，为什么不能编译这些代码呢？

不能进行编译的原因是不能把 **SmartPtr<CD>** 或 **SmartPtr<Cassette>** 转换成 **SmartPtr<MusicProduct>**。从编译器的观点来看，这些类之间没有任何关系。为什么编译器的会这样认为呢？毕竟 **SmartPtr<CD>** 或 **SmartPtr<Cassette>** 不是从 **SmartPtr<MusicProduct>**继承过来的，这些类之间没有继承关系，我们不可能要求编译器把一种对象转换成（完全不同的）另一种类型的对象。

幸运的是，有办法避开这种限制，这种方法的核心思想（不是实际操作）很简单：对于可以进行隐式转换的每个灵巧指针类型都提供一个隐式类型转换操作符（参见条款 **M5**）。例如在 **music** 类层次内，在 **Cassette** 和 **CD** 的灵巧指针类内你可以加入 **operator SmartPtr<MusicProduct>**函数：

```
class SmartPtr<Cassette> {
public:
  operator SmartPtr<MusicProduct>()
  { return SmartPtr<MusicProduct>(pointee); }
  ...
private:
  Cassette *pointee;
};
class SmartPtr<CD> {
public:
  operator SmartPtr<MusicProduct>()
  { return SmartPtr<MusicProduct>(pointee); }
  ...
private:
  CD *pointee;
};
```

这种方法有两个缺点。第一，你必须人为地特化（**specialize**）**SmartPtr** 类，所以你加入隐式类型转换操作符也就破坏了模板的通用性。第二，你可能必须添加许多类型转换符，因为你指向的对象可以位于继承层次中很深的位置，你必须为直接或间接继承的每一个基类提供一个类型转换符。（如果你认为你能够克服这个缺点，方法是仅仅为转换到直接基类而提供一个隐式类型转换符，那么请再想想这样做行么？因为编译器在同一时间调用用户定义

类型转换函数的次数不能超过一次，它们不能把指向 T 的灵巧指针转换为指向 T 的间接基类的灵巧指针，除非只要一步就能完成。）

如果你能让编译器为你编写所有的类型转换函数，这会节省很多时间。感谢最近的语言扩展，让你能够做到，这个扩展能声明（非虚）成员函数模板（通常就叫成员模板（member template）），你能使用它来生成灵巧指针类型转换函数，如下：

```
template<class T>                    // 模板类，指向 T 的
class SmartPtr {                     // 灵巧指针
public:
  SmartPtr(T* realPtr = 0);
  T* operator->() const;
  T& operator*() const;
  template<class newType>             // 模板成员函数
  operator SmartPtr<newType>()        // 为了实现隐式类型转换.
  {
    return SmartPtr<newType>(pointee);
  }
  ...
};
```

现在请你注意，这可不是魔术——不过也很接近于魔术。它的原理如下所示。（如果下面的内容让你感到既冗长又令你费解，请不要失望，一会儿我会给出一个例子。我保证你看完例子后，就能够更深入地理解这段内容了。）假设编译器有一个指向 T 对象的灵巧指针，它要把这个对象转换成指向“T 的基类”的灵巧指针。编译器首先检查 SmartPtr<T>的类定义，看其有没有声明明确的类型转换符，但是它没有声明。（这不是指：在上面的模板没有声明类型转换符。）编译器然后检查是否存在一个成员函数模板，并可以被实例化成它所期望的类型转换。它发现了一个这样的模板（带有形式类型参数 newType），所以它把 newType 绑定成 T 的基类类型来实例化模板。这时，惟一的问题是实例化的成员函数代码能否被编译：传递（dumb）指针 pointee 到指向“T 的基类”的灵巧指针的构造函数，必须合法的。指针 pointee 是指向 T 类型的，把它转变成指向其基类（public 或 protected）对象的指针必然是合法的，因此类型转换操作符能够被编译，可以成功地把指向 T 的灵巧指针隐式地类型转换为指向“T 的基类”的灵巧指针。

举一个例子会有所帮助。让我们回到 CDs、cassettes、music 产品的继承层次上来。我们先前已经知道下面这段代码不能被编译，因为编译器不能把指向 CD 的灵巧指针转换为指向 music 产品的灵巧指针：

```
void displayAndPlay(const SmartPtr<MusicProduct>& pmp,
                    int howMany);
SmartPtr<Cassette> funMusic(new Cassette("Alapalooza"));
SmartPtr<CD> nightmareMusic(new CD("Disco Hits of the 70s"));
displayAndPlay(funMusic, 10);          // 以前是一个错误
displayAndPlay(nightmareMusic, 0);     // 以前是一个错误
```

修改了灵巧指针类，包含了隐式类型转换操作符的成员函数模板以后，这个代码就可以成功运行了。拿如下调用举例，看看为什么能够成功运行：

```
displayAndPlay(funMusic, 10);
```

funMusic 对象的类型是 SmartPtr<Cassette>。函数 displayAndPlay 期望的参数是 SmartPtr<MusicProduct>地对象。编译器侦测到类型不匹配，于是寻找把 funMusic 转换成 SmartPtr<MusicProduct>对象的方法。它在 SmartPtr<MusicProduct>类里寻找带有 SmartPtr<Cassette>类型参数的单参数构造函数（参见条款 M5），但是没有找到。然后它们又寻找成员函数模板，以实例化产生这样的函数。它们在 SmartPtr<Cassette>发现了模板，把 newType 绑定到 MusicProduct 上，生成了所需的函数。实例化函数，生成这样的代码：

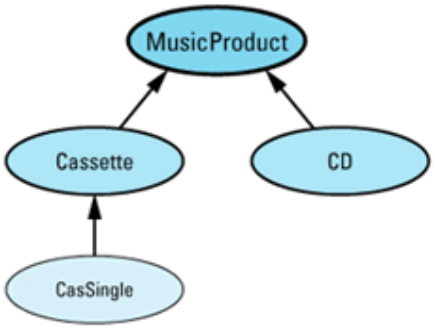
```
SmartPtr<Cassette>::  operator SmartPtr<MusicProduct>()
{
  return SmartPtr<MusicProduct>(pointee);
```

```
}
```

能编译这行代码么？实际上这段代码就是用 **pointee** 做为参数调用 **SmartPtr<MusicProduct>**的构造函数，所以真正的问题是能否用一个 **Cassette***指针构造一个 **SmartPtr<MusicProduct>**对象。现在我们对 **dumb** 指针类型之间的转换已经很熟悉了，**Cassette***能够被传递给需要 **MusicProduct***指针的地方。因此 **SmartPtr<MusicProduct>**构造函数可以成功调用，同样 **SmartPtr<Cassette>**到 **SmartPtr<MusicProduct>**之间的类型转换也能成功进行。太棒了，实现了灵巧指针之间的类型转换，还有什么比这更简单么？

此外，还有其它功能么？不要被这个例子误导，而认为这种方法只能用于把指针在继承层次中向上进行类型转换。这种方法可以成功地用于任何合法的指针类型转换。如果你有 **dumb** 指针 **T1***和另一种 **dumb** 指针 **T2***，当且仅当你能隐式地把 **T1***转换为 **T2***时，你就能够隐式地把指向 **T1** 的灵巧指针类型转换为指向 **T2** 的灵巧指针类型。

这种技术能给我们几乎所有想要的行为特性。假设我们用一个新类 **CasSingle** 来扩充 **MusicProduct** 类层次，用来表示 **cassette singles**。修改后的类层次看起来象这样：



现在考虑这段代码：

```
template<class T>                       // 同上，包括作为类型
class SmartPtr { ... };                 // 转换操作符的成员模板
void displayAndPlay(const SmartPtr<MusicProduct>& pmp,
                    int howMany);
void displayAndPlay(const SmartPtr<Cassette>& pc,
                    int howMany);
SmartPtr<CasSingle> dumbMusic(new CasSingle("Achy Breaky Heart"));
displayAndPlay(dumbMusic, 1);           // 错误!
```

在这个例子里，**displayAndPlay** 被重载，一个函数带有 **SmartPtr<Cassette>** 对象参数，其它函数的参数为 **SmartPtr<CasSingle>**，我们期望调用 **SmartPtr<Cassette>**，因为 **CasSingle** 是直接从 **Cassette** 上继承下来的，而间接继承自 **MusicProduct**。当然这是 **dumb** 指针时的工作方法。我们的灵巧指针不会这么灵巧，它们的转换操作符是成员函数，对 **C++** 编译器而言，所有类型转换操作符是同等地位的。因此 **displayAndPlay** 的调用具有二义性，因为从 **SmartPtr<CasSingle>** 到 **SmartPtr<Cassette>** 的类型转换并不比到 **SmartPtr<MusicProduct>**的类型转换优先。

通过成员模板来实现灵巧指针的类型转换还有两个缺点。第一，支持成员模板的编译器较少，所以这种技术不具有可移植性。以后情况会有所改观，但是没有人知道这会等到什么时候。第二，这种方法的工作原理不很明了，要理解它必须先要深入理解函数调用时的参数匹配，隐式类型转换函数，模板函数隐式实例化和成员函数模板。有些程序员以前从来没有看到过这种技巧，而却被要求维护使用了这种技巧的代码，我真是很可怜他们。这种技巧确实很巧妙，这自然是肯定，但是过于的巧妙可是一件危险的事情。

不要再拐弯抹角了，直接了当地说，我们想要知道的是在继承类向基类进行类型转换方面，我们如何能够让灵巧指针的行为与 **dumb** 指针一样呢？答案很简单：不可能。正如 **Daniel Edelson** 所说，灵巧指针固然灵巧，但不是指针。最好的方法是使用成员模板生成类型转换

函数，在会产生二义性结果的地方使用 **casts**（类型转换，参见条款 **M2**）。这不是一个完美的方法，不过已经很不错了，在一些情况下需去除二义性，所付出的代价与灵巧指针提供复杂的功能相比还是值得的。

I 灵巧指针和 const

对于 **dumb** 指针来说，**const** 既可以针对指针所指向的东西，也可以针对于指针本身，或者兼有两者的含义（参见 **Effective C++**条款 **21**）：

```
CD goodCD("Flood");
const CD *p;                          // p 是一个 non-const 指针
                                      //指向 const CD 对象
CD * const p = &goodCD;               // p 是一个 const 指针
                                      // 指向 non-const CD 对象;
                                      // 因为 p 是 const, 它
                                      // 必须被初始化
const CD * const p = &goodCD;         // p 是一个 const 指针
                                      // 指向一个 const CD 对象
```

我们自然想要让灵巧指针具有同样的灵活性。不幸的是只能在一个地方放置 **const**，并只能对指针本身起作用，而不能针对于所指对象：

```
const SmartPtr<CD> p =                 // p 是一个 const 灵巧指针
  &goodCD;                              // 指向 non-const CD 对象
```

好像有一个简单的补救方法，就是建立一个指向 **cosnt CD** 的灵巧指针：

```
SmartPtr<const CD> p =              // p 是一个 non-const 灵巧指针
  &goodCD;                          // 指向 const CD 对象
```

现在我们可以建立 **const** 和 **non-const** 对象和指针的四种不同组合：

```
SmartPtr<CD> p;                           // non-const 对象
                                          // non-const 指针
SmartPtr<const CD> p;                     // const 对象,
                                          // non-const 指针
const SmartPtr<CD> p = &goodCD;           // non-const 对象
                                          // const 指针
const SmartPtr<const CD> p = &goodCD;     // const 对象
                                          // const 指针
```

但是美中不足的是，使用 **dumb** 指针我们能够用 **non-const** 指针初始化 **const** 指针，我们也能用指向 **non-cosnt** 对象的指针初始化指向 **const** 对象的指针；就像进行赋值一样。例如：

```
CD *pCD = new CD("Famous Movie Themes");
const CD * pConstCD = pCD;                // 正确
```

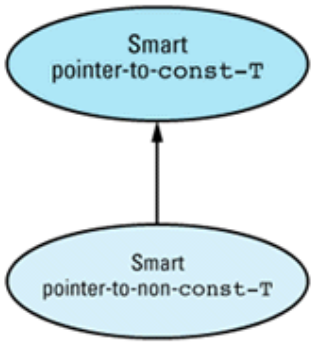
但是如果我们试图把这种方法用在灵巧指针上，情况会怎么样呢？

```
SmartPtr<CD> pCD = new CD("Famous Movie Themes");
SmartPtr<const CD> pConstCD = pCD;        // 正确么?
```

SmartPtr<CD> 与 **SmartPtr<const CD>**是完全不同的类型。在编译器看来，它们是毫不相关的，所以没有理由相信它们是赋值兼容的。到目前为止这是一个老问题了，把它们变成赋值兼容的唯一方法是你必须提供函数，用来把 **SmartPtr<CD>**类型的对象转换成 **SmartPtr<const CD>**类型。如果你使用的编译器支持成员模板，就可以利用前面所说的技巧自动生成你需要的隐式类型转换操作。（我前面说过，只要对应的 **dumb** 指针能进行类型转换，灵巧指针也就能进行类型转换，我没有欺骗你们。带 **const** 的类型转换也没有问题。）如果你没有这样的编译器，你必须克服更大的困难。

带 **const** 的类型转换是单向的：从 **non-const** 到 **const** 的转换是安全的，但是从 **const** 到 **non-const** 则不是安全的。而且用 **const** 指针能做的事情，用 **non-const** 指针也能做，但是用 **non-const** 指针还能做其它一些事情（例如，赋值操作）。同样，用指向 **const** 对象的指针能做的任何事情，用指向 **non-const** 对象的指针也能做到，但是用指向 **non-const** 对象的指针能够完成一些指向 **const** 对象的指针所不能完成的事情（例如，赋值操作）。

这些规则看起来与 **public** 继承的规则相类似（**Effective C++** 条款 **35**）。你能够把一个派生类对象转换成基类对象，但是反之则不是这样，你对基类所做的任何事情对派生类也能做，但是还能对派生类做另外一些事情。我们能够利用这一点来实现灵巧指针，就是说可以让每个指向 **T** 的灵巧指针类 **public** 派生自一个对应的指向 **const-T** 的灵巧指针类：




```
template<class T>                    // 指向 const 对象的
class SmartPtrToConst {              // 灵巧指针
  ...                                // 灵巧指针通常的
                                     // 成员函数
protected:
  union {
    const T* constPointee;           // 让 SmartPtrToConst 访问
    T* pointee;                      // 让 SmartPtr 访问
  };
};
template<class T>                    // 指向 non-const 对象
class SmartPtr:                      // 的灵巧指针
  public SmartPtrToConst<T> {
  ...                                // 没有数据成员
};
```

使用这种设计方法，指向 **non-const-T** 对象的灵巧指针包含一个指向 **const-T** 的 **dumb** 指针，指向 **const-T** 的灵巧指针需要包含一个指向 **cosnt-T** 的 **dumb** 指针。最方便的方法是把指向 **const-T** 的 **dumb** 指针放在基类里，把指向 **non-const-T** 的 **dumb** 指针放在派生类里，然而这样做有些浪费，因为 **SmartPtr** 对象包含两个 **dumb** 指针**:**一个是从 **SmartPtrToConst** 继承来的，一个是 **SmartPtr** 自己的。

一种在 **C** 世界里的老式武器可以解决这个问题，这就是 **union**，它在 **C++**中同样有用。**Union** 在 **protected** 中，所以两个类都可以访问它，它包含两个必须的 **dumb** 指针类型，**SmartPtrToConst<T>**对象使用 **constPointee** 指针，**SmartPtr<T>**对象使用 **pointee** 指针。因此我们可以在不分配额外空间的情况下，使用两个不同的指针（参见 **Effective C++**条款 **10** 中另外一个例子）这就是 **union** 美丽的地方。当然两个类的成员函数必须约束它们自己仅仅使用适合的指针。这是使用 **union** 所冒的风险。

利用这种新设计，我们能够获得所要的行为特性：

```
SmartPtr<CD> pCD = new CD("Famous Movie Themes");
SmartPtrToConst<CD> pConstCD = pCD;      // 正确
```

I 评价

有关灵巧指针的讨论该结束了，在我们离开这个话题之前，应该问这样一个问题：灵巧指针如此繁琐麻烦，是否值得使用，特别是如果你的编译器缺乏支持成员函数模板时。

通常是值得的。例如通过使用灵巧指针极大地简化了条款 **M29** 中的引用计数代码。而且

正如该例子所显示的，灵巧指针的使用在一些领域受到极大的限制，例如测试空值、转换到 dumb 指针、继承类向基类转换和对指向 const 的指针的支持。同时灵巧指针的实现、理解和维护需要大量的技巧。调试使用灵巧指针的代码也比调试使用 dumb 指针的代码困难。无论如何你也不可能设计出一种通用目的的灵巧指针，能够替代 dumb 指针。

达到同样的代码效果，使用灵巧指针更方便。灵巧指针应该谨慎使用，不过每个 C++程序员最终都会发现它们是有用的。

7.5 Item M29：引用计数

引用计数是这样一个技巧，它允许多个有相同值的对象共享这个值的实现。这个技巧有两个常用动机。第一个是简化跟踪堆中的对象的过程。一旦一个对象通过调用 new 被分配出来，最要紧的就是记录谁拥有这个对象，因为其所有者——并且只有其所有者——负责对这个对象调用 delete。但是，所有权可以被从一个对象传递到另外一个对象（例如通过传递指针型参数），所以跟踪一个对象的所有权是很困难的。象 auto_ptr（见 Item M9）这样的类可以帮助我们，但经验显示大部分程序还不能正确地得到这样的类。引用计数可以免除跟踪对象所有权的担子，因为当使用引用计数后，对象自己拥有自己。当没人再使用它时，它自己自动销毁自己。因此，引用计数是个简单的垃圾回收体系。

第二个动机是由于一个简单的常识。如果很多对象有相同的值，将这个值存储多次是很无聊的。更好的办法是让所有的对象共享这个值的实现。这么做不但节省内存，而且可以使得程序运行更快，因为不需要构造和析构这个值的拷贝。

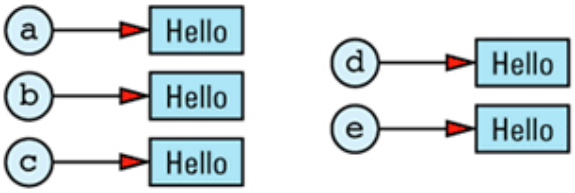
和大部分看似简单的主意一样，这个动机也有一个曲折而有趣的细节。在其中必须有一个正确实现的引用计数体系。在开始钻研细节前，让我们掌握一些基础。一个好主意是先着眼于我们将可能如何遇到多个对象有相同的值。这儿有一个：

```
class String {                      // the standard string type may
public:                             // employ the techniques in this
                                    // Item, but that is not required
  String(const char *value = "");
  String& operator=(const String& rhs);
  ...
private:
  char *data;
};
String a, b, c, d, e;
a = b = c = d = e = "Hello";
```

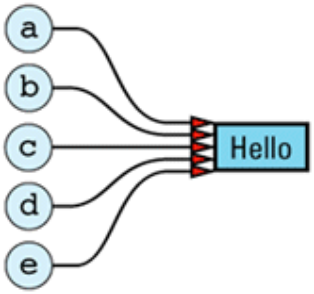
看起来，对象 a 到 e 都有相同的值“Hello”。其值的形态取决于 String 类是怎么实现的，但通常的实现是每个 string 对象有一个这个值的拷贝。例如，String 的赋值操作可能实现为这样：

```
String& String::operator=(const String& rhs)
{
  if (this == &rhs) return *this;         // see Item E17
  delete [] data;
  data =   new char[strlen(rhs.data) + 1];
  strcpy(data, rhs.data);
  return *this;                           // see Item E15
}
```

根据这个实现，我们可以推测，这 **5** 个对象及其值如下：



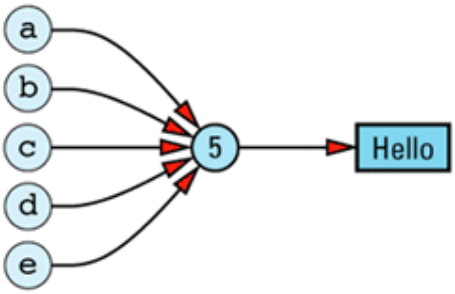
其冗余是显然的。在一个理想的世界中，我们希望将上图改为这样：



这里，只存储了一个“**Hello**”的拷贝，所有具有此值的 **String** 对象共享其实现。

实际世界中，实现这个主意是不可能的，因为我们需要跟踪多少对象共享同一个值。如果上面的对象 **a** 被赋了“**Hello**”以外的另外一个值，我们不能摧毁值“**Hello**”，因为还有四个对象需要它。另一方面，如果只有一个对象有“**Hello**”这个值，当其超出生存空间时，没有对象具有这个值了，我们必须销毁这个值以避免资源泄漏。

保存当前共享/引用同一个值的对象数目的需求意味着我们的那张图必须增加一个计数值（引用计数）：



（有些人将其叫作 **use count**，但我不是其中之一。**C++**有很多它自己的特性，最后需要的一个是专业名词的派别之争。）

I 实现引用计数

创建一个带引用计数的 **String** 类并不困难，但需要注意一些细节，所以我们将略述这样一个类的大部分常用成员函数的实现。然而，在开始之前，认识到“我们需要一个地方来存储这个计数值”是很重要的。这个地方不能在 **String** 对象内部，因为需要的是每个 **String** 值一个引用计数值，而不是每个 **String** 对象一个引用计数。这意味着 **String** 值和引用计数间是一一对应的关系，所以我们将创建一个类来保存引用计数及其跟踪的值。我们叫这个类 **StringValue**，又因为它唯一的用处就是帮助我们实现 **String** 类，所以我们将它嵌套在 **String** 类的私有区内。另外，为了便于 **Sting** 的所有成员函数读取其数据区，我们将 **StringValue** 申明为 **struct**。需要知道的是：将一个 **struct** 内嵌在类的私有区内，能便于这个类的所有成员访问这个结构，但阻止了其它任何人对它的访问（当然，除了友元）。

基本设计是这样的：

```
class String {
public:
  ...                              // the usual String member
                                   // functions go here
private:
  struct StringValue { ... };      // holds a reference count
                                   // and a string value
  StringValue *value;              // value of this String
};
```

我们可以给这个类起个其它名字（如 **RCString**）以强调它使用了引用计数，但类的实现不该是类的用户必须关心的东西，用户只关心类的公有接口。而我们带引用计数的 **String** 版本与不带引用计数的版本，其接口完全相同，所以为什么要用类的名字来把问题搅混呢？真的需要吗？所以我们没有这么做。

这是 StringValue 的实现：

```
class String {
private:
    struct StringValue {
      int refCount;
      char *data;
      StringValue(const char *initValue);
      ~StringValue();
    };
    ...
};
String::StringValue::StringValue(const char *initValue)
: refCount(1)
{
  data = new char[strlen(initValue) + 1];
  strcpy(data, initValue);
}
String::StringValue::~StringValue()
{
  delete [] data;
}
```

这是其所有的一切，很清楚，这不足以实现带引用计数的 String 类。一则，没有拷贝构造函数和赋值运算（见 Item E11）；二则，没有提供对 refCount 的操作。别担心，少掉的功能将由 String 类提供。StringValue 的主要目的是提供一个空间将一个特别的值和共享此值的对象的数目联系起来。StringValue 给了我们这个，这就足够了。

我们现在开始处理 String 的成员函数。首先是构造函数：

```
class String {
public:
  String(const char *initValue = "");
  String(const String& rhs);
  ...
};
```

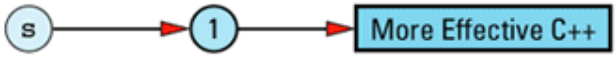
第一个构造函数被实现得尽可能简单。我们用传入的 char *字符串创建了一个新的 StringValue 对象，并将我们正在构造的 string 对象指向这个新生成的 StringValue：

```
String::String(const char *initValue)
: value(new StringValue(initValue))
{}
```

这样的用户代码：

```
String s("More Effective C++");
```

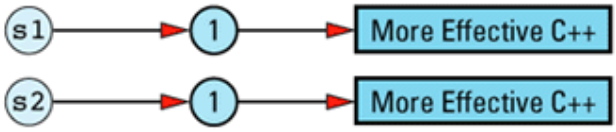
生成的数据结构是这样的：



String 对象是独立构造的，有同样初始化值的对象并不共享数据，所以，这样的用户代码：

```
String s1("More Effective C++");
String s2("More Effective C++");
```

产生这样的数据结构：



消除这样的副本是可能的：通过让 **String**（或 **StringValue**）对象跟踪已存在的 **StringValue** 对象，并只在是不同串时才创建新的对象。但这样的改进有些偏离目标。于是，我将它作为习题留给读者。

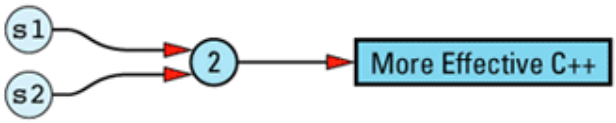
String 的拷贝构造函数很高效：新生成的 **String** 对象与被拷贝的对象共享相同的 **StringValue** 对象：

```
String::String(const String& rhs)
: value(rhs.value)
{
  ++value->refCount;
}
```

这样的代码：

```
String s1("More Effective C++");
String s2 = s1;
```

产生这样的数据结构：



这肯定比通常的（不带引用计数的）**string** 类高效，因为不需要为新生成的 **string** 值分配内存、释放内存以及将内容拷贝入这块内存。现在，我们只不过是拷贝了一个指针并增加了一次引用计数。

String 类的析构函数同样容易实现，因为大部分情况下它不需要做任何事情。只要引用计数值不是 **0**，也就是至少有一个 **String** 对象使用这个值，这个值就不可以被销毁。只有当唯一的使用者被析构了（也就是引用计数在进入函数前已经为 **1** 时），**String** 的析构函数才摧毁 **StringValue** 对象：

```
class String {
public:
  ~String();
  ...
};
String::~String()
{
  if (--value->refCount == 0) delete value;
}
```

和没有引用计数的版本比较一下效率。那样的函数总调用 **delete**，当然会有一个相当程度的运行时间的代价。现在提供的 **String** 对象们实际上有时具有相同的值，上面的这个实现在此时只需要做一下减少引用计数并与 **0** 进行比较。

如果在这个问题上引用计数没有向外界表现出来，你就根本不需要花注意力。

这就是 **String** 的构造和析构，我们现在转到赋值操作：

```
class String {
public:
  String& operator=(const String& rhs);
  ...
};
```

当用户写下这样的代码：

```
s1 = s2;                              // s1 and s2 are both String objects
```

其结果应该是 **s1** 和 **s2** 指向相同的 **StringValue** 对象。对象的引用计数应该在赋值时被增加。并且，**s1** 原来指向的 **StringValue** 对象的引用计数应该减少，因为 **s1** 不再具有这

个值了。如果 s1 是拥有原来的值的唯一对象，这个值应该被销毁。在 C++中，其实现看起来是这样的：

```
String& String::operator=(const String& rhs)
{
  if (value == rhs.value) {          // do nothing if the values
    return *this;                    // are already the same; this
  }                                  // subsumes the usual test of
                                     // this against &rhs (see Item E17)
  if (--value->refCount == 0) {      // destroy *this's value if
    delete value;                    // no one else is using it
  }
  value = rhs.value;                 // have *this share rhs's
  ++value->refCount;                 // value
  return *this;
}
```

写时拷贝

围绕我们的带引用计数的 String 类，考虑一下数组下标操作（[]），它允许字符串中的单个字符被读或写：

```
class String {
public:
  const char&
    operator[](int index) const;       // for const Strings
  char& operator[](int index);           // for non-const Strings
...
};
```

这个函数的 const 版本的实现很容易，因为它是一个只读操作，String 对象的值不受影响：

```
const char& String::operator[](int index) const
{
  return value->data[index];
}
```

（这个函数实现了 C++传统意义上的下标索引（根本不会说“不”）。如果你想加上参数检查，这是非常容易的。）

非 const 的 operator[]版本就是一个完全不同的故事了。它可能是被调用了来读一个

字符，也可能被调用了来写一个字符：

```
String s;
...
cout << s[3];                    // this is a read
s[5] = 'x';                      // this is a write
```

我们希望以不同的方式处理读和写。简单的读操作，可以用与 **const** 的 **operator[]**类似的方式实现，而写操作必须用完全不同的方式来实现。

当我们修改一个 **String** 对象的值时，必须小心防止修改了与它共享相同 **StringValue** 对象的其它 **String** 对象的值。不幸的是，**C++**编译器没有办法告诉我们一个特定的 **operator[]** 是用作读的还是写的，所以我们必须保守地假设“所有”调用非 **const operator[]**的行为都是为了写操作。（**Proxy** 类可以帮助我们区分读还是写，见 **Item M30**。）

为了安全地实现非 **const** 的 **operator[]**，我们必须确保没有其它 **String** 对象在共享这个可能被修改的 **StringValue** 对象。简而言之，当我们返回 **StringValue** 对象中的一个字符的引用时，必须确保这个 **StringValue** 的引用计数是 **1**。这儿是我们的实现：

```
char& String::operator[](int index)
{
  // if we're sharing a value with other String objects,
  // break off a separate copy of the value for ourselves
  if (value->refCount > 1) {
    --value->refCount;                  // decrement current value's
                                        // refCount, because we won't
                                        // be using that value any more
    value =                             // make a copy of the
      new StringValue(value->data);     // value for ourselves
  }
  // return a reference to a character inside our
  // unshared StringValue object
  return value->data[index];
}
```

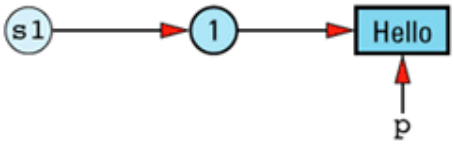
这个“与其它对象共享一个值直到写操作时才拥有自己的拷贝”的想法在计算机科学中已经有了悠久而著名的历史了，尤其是在操作系统中：进程共享内存页直到它们想在自己的页拷贝中修改数据为止。这个技巧如此常用，以至于有一个名字：写时拷贝。它是提高效率的一个更通用方法——**Lazy** 原则——的特例。

I 指针、引用与写时拷贝

大部分情况下，写时拷贝可以同时保证效率和正确性。只有一个挥之不去的问题。看一下这样的代码：

```
String s1 = "Hello";
char *p = &s1[1];
```

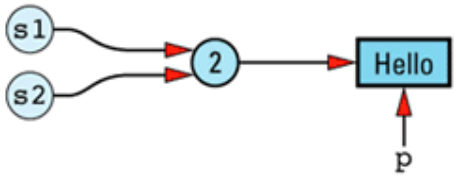
数据结构是这样的：



现在看增加一条语句：

```
String s2 = s1;
```

String 的拷贝构造函数使得 **s2** 共享 **s1** 的 **StringValue** 对象，所以数据结构将是：



下面这样的语句将有不受欢迎的结果：

```
*p = 'x';                       // modifies both s1 and s2!
```

String 的拷贝构造函数没有办法检测这样的问题，因为它不知道指向 **s1** 拥有的 **StringValue** 对象的指针的存在。并且，这个问题不局限于指针：它同样存在于有人保存了一个 **String** 的非 **const operator[]**的返回值的引用的情况下。

至少有三种方法来应付这个问题。第一个是忽略它，假装它不存在。这是实现带引用计数的 **String** 类的类库中令人痛苦的常见问题。如果你有带引用计数的 **String** 类，试一下上面的例子，看你是否很痛苦。即使你不能确定你操作的是否是带引用计数的 **String** 类，也无论如何应该试一下这个例子。由于封装，你可能使用了一个这样的类型而不自知。

不是所以的实现都忽略这个问题。稍微好些的方法是明确说明它的存在。通常是将它写入文档，或多或少地说明“别这么做。如果你这么做了，结果为未定义。”无论你以哪种方式这么做了（有意地或无意地），并抱怨其结果时，他们辩解道：“好了，我们告诉过你别这么做的。”这样的实现通常很方便，但它们在可用性方面留下了太多的期望。

第三个方法是排除这个问题。它不难实现，但它将降低一个值共享于对象间的次数。它的本质是这样的：在每个 **StringValue** 对象中增加一个标志以指出它是否为可共享的。在最初（对象可共享时）将标志打开，在非 **const** 的 **operator[]**被调用时将它关闭。一旦标志被设为 **false**，它将永远保持在这个状态（注 **10**）。

这是增加了共享标志的修改版本：

```
class String {
private:
  struct StringValue {
    int refCount;
    bool shareable;                // add this
    char *data;
    StringValue(const char *initValue);
    ~StringValue();
  };
...
};
String::StringValue::StringValue(const char *initValue)
:   refCount(1),
    shareable(true)                // add this
{
  data = new char[strlen(initValue) + 1];
  strcpy(data, initValue);
}
String::StringValue::~StringValue()
{
  delete [] data;
}
```

如你所见，并不需要太多的改变；需要修改的两行都有注释。当然，**String** 的成员函数也必须被修改以处理这个共享标志。这里是拷贝构造函数的实现：

```
String::String(const String& rhs)
{
  if (rhs.value->shareable) {
    value = rhs.value;
    ++value->refCount;
  }
  else {
    value = new StringValue(rhs.value->data);
  }
```

```
}
```

所有其它的成员函数也都必须以类似的方法检查这个共享标志。非 **const** 的 **operator[]** 版本是唯一将共享标志设为 **false** 的地方：

```
char& String::operator[](int index)
{
  if (value->refCount > 1) {
    --value->refCount;
    value = new StringValue(value->data);
  }
  value->shareable = false;           // add this
  return value->data[index];
}
```

如果使用 **Item M30** 中的 **proxy** 类的技巧以区分读写操作，你通常可以降低必须被设为不可共享的 **StringValue** 对象的数目。

I 带引用计数的基类

引用计数不只用在字符串类上，只要是多个对象具有相同值的类都可以使用引用计数。改写一个类以获得引用计数需要大量的工作，而我们已经有太的工作需要做了。这样不好吗：如果我们将引用计数的代码写成与运行环境无关的，并能在需要时将它嫁接到其它类上？当然很好。很幸运，有一个方法可以实现它（至少完成了绝大部分必须的工作）。

第一步是构建一个基类 **RCObject**，任何需要引用计数的类都必须从它继承。**RCObject** 封装了引用计数功能，如增加和减少引用计数的函数。它还包含了当这个值不再被需要时摧毁值对象的代码（也就是引用计数为 **0** 时）。最后，它包含了一个字段以跟踪这个值对象是否可共享，并提供查询这个值和将它设为 **false** 的函数。不需将可共享标志设为 **true** 的函数，因为所有的值对象默认都是可共享的。如上面说过的，一旦一个对象变成了不可共享，将没有办法使它再次成为可共享。

RCObject 的定义如下：

```
class RCObject {
public:
  RCObject();
  RCObject(const RCObject& rhs);
  RCObject& operator=(const RCObject& rhs);
  virtual ~RCObject() = 0;
  void addReference();
  void removeReference();
```

```
void markUnshareable();
bool isShareable() const;
bool isShared() const;
private:
int refCount;
bool shareable;
};
```

RCObjcet 可以被构造（作为派生类的基类部分）和析构；可以有新的引用加在上面以及移除当前引用；其可共享性可以被查询以及被禁止；它们可以报告当前是否被共享了。这就是它所提供的功能。对于想有引用计数的类，这确实就是我们所期望它们完成的东西。注意虚析构函数，它明确表明这个类是被设计了作基类使用的（见 **Item E14**）。同时要注意这个析构函数是纯虚的，它明确表明这个类只能作基类使用。

RCOject 的实现代码：

```
RCObject::RCObject()
: refCount(0), shareable(true) {}
RCObject::RCObject(const RCObject&)
: refCount(0), shareable(true) {}
>RCObject& RCObject::operator=(const RCObject&)
{ return *this; }
RCObject::~RCObject() {}              // virtual dtors must always
                                      // be implemented, even if
                                      // they are pure virtual
                                      // and do nothing (see also
                                      // Item M33 and Item E14)
void RCObject::addReference() { ++refCount; }
void RCObject::removeReference()
{   if (--refCount == 0) delete this; }
void RCObject::markUnshareable()
{ shareable = false; }
bool RCObject::isShareable() const
{ return shareable; }
bool RCObject::isShared() const
{ return refCount > 1; }
```

可能很奇怪，我们在所有的构造函数中都将 **refCount** 设为了 **0**。这看起来违反直觉。

确实，最少，构造这个 **RCObject** 对象的对象引用它！在它构造后，只需构造它的对象简单地将 **refCount** 设为 **1** 就可以了，所以我们没有将这个工作放入 **RCObject** 内部。这使得最终的代码看起来很简短。

另一个奇怪之处是拷贝构造函数也将 **refCount** 设为 **0**，而不管被拷贝的 **RCObject** 对象的 **refCount** 的值。这是因为我们正在构造新的值对象，而这个新的值对象总是未被共享的，只被它的构造者引用。再一次，构造者负责将 **refCount** 设为正确的值。

RCObject 的赋值运算看起来完全出乎意料：它没有做任何事情。这个函数不太可能被调用的。**RCObject** 是基于引用计数来共享的值对象的基类，它不该被从一个赋给另外一个，而应该是拥有这个值的对象被从一个赋给另外一个。在我们这个设计里，我们不期望 **StringValue** 对象被从一个赋给另外一个，我们期望在赋值过程中只有 **String** 对象被涉及。在 **String** 参与的赋值语句中，**StringValue** 的值没有发生变化，只是它的引用计数被修改了。

不过，可以想象，一些还没有写出来的类在将来某天可能从 **RCObject** 派生出来，并希望允许被引用计数的值被赋值（见 **Item M23** 和 **Item E16**）。如果这样的话，**RCObject** 的赋值操作应该做正确的事情，而这个正确的事情就是什么都不做。想清楚了吗？假设我们希望允许在 **StringValue** 对象间赋值。对于给定的 **StringValue** 对象 **sv1** 和 **sv2**，在赋值过程中，它们的引用计数值上发生什么？

```
sv1 = sv2;                    // how are sv1's and sv2's reference
                              // counts affected?
```

在赋值之前，已经有一定数目的 **String** 对象指向 **sv1**。这个值在赋值过程中没有被改变，因为只是 **sv1** 的值被改变了。同样的，一定数目的 **String** 对象在赋值之前指向前 **v2**，在赋值后，同样数目的对象指向 **sv2**。**sv2** 的引用计数同样没有改变。当 **RCObject** 在赋值过程中被涉及时，指向它的对象的数目没有受影响，因此 **RCObject::operator=**不应该改变引用计数值。上面的实现是正确的。违反直觉？可能吧，但它是正确的。

RCObject::removeReference 的代码不但负责减少对象的 **refCount** 值，还负责当 **refCount** 值降到 **0** 时析构对象。后者是通过 **delete this** 来实现的，如 **Item M27** 中解释的，这只当我们知道***this** 是一个堆对象时才安全。要让这个类正确，我们必须确保 **RCObject** 只能被构建在堆中。实现这一点的常用方法见 **Item M27**，但我们这次采用一个特别的方法，这将在本条款最后讨论。

为了使用我们新写的引用计数基类，我们将 **StringValue** 修改为是从 **RCObject** 继承而得到引用计数功能的：

```
class String {
private:
  struct StringValue: public RCObject {
```

```
    char *data;
    StringValue(const char *initValue);
    ~StringValue();
  };
...
};
String::StringValue::StringValue(const char *initValue)
{
  data = new char[strlen(initValue) + 1];
  strcpy(data, initValue);
}
String::StringValue::~StringValue()
{
  delete [] data;
}
```

这个版本的 **StringValue** 和前面的几乎一样，唯一改变的就是 **StringValue** 的成员函数不再处理 **refCount** 字段。**RCObject** 现在接管了这个工作。

不用感觉不舒服，如果你注意到嵌套类（**StringValue**）从一个与包容类（**String**）无关的类（**RCObject**）继承而来的话。它第一眼看上去是有些古怪，但完全合理。嵌套类和其它类是完全相同的，所以它有自由从它喜欢的任何其它类继承。以后，你不用第二次思考这种继承关系了。

I 自动的引用计数处理

RCObject 类给了我们一个存储引用计数的地方，并提供了成员函数供我们操作引用计数，但调用这些函数的动作还必须被手工加入其它类中。仍然需要在 **String** 的拷贝构造函数和赋值运算函数中调用 **StringValue** 的 **addReference** 和 **removeReference** 函数。这很笨拙。我们想将这些调用也移入一个可重用的类中，以使得 **String** 这样的类的作者不用再担心引用计数的任何细节。能实现吗？**C++**支持这样的重用吗？

能。没有一个简单的方法将所有引用计数方面的工作从所有的类中移出来；但有一个方法可以从大部分类中将大部分工作移出来。（在一些类中，你可以消除所有引用计数方面的代码，但我们的 **String** 类不是其中之一。有一个成员函数搞坏了这件事，我希望你别吃惊，它是我们的老对头：非 **const** 版本的 **operator[]**。别放心上，我们最终制服了这家伙。）

每个 **String** 对象包含一个指针指向 **StringValue** 对象：

```
class String {
private:
```

```
  struct StringValue: public RCObject { ... };
  StringValue *value;                // value of this String
  ...
};
```

我们必须操作 **StringValue** 对象的 **refCount** 字段，只要任何时候任一个指向它的指针身上发生了任何有趣的事件。“有趣的事件”包括拷贝指针、给指针赋值和销毁指针。如果我们能够让指针自己检测这些事件并自动地执行对 **refCount** 字段的必须操作，那么我们就自由了。不幸的是，指针功能很弱，对任何事情作检测并作出反应都是不可能的。还好，有一个办法来增强它们：用行为类似指针的对象替代它们，但那样要多做很多工作了。

这样的对象叫灵巧指针，你可以在 **Item M28** 这看到它的更多细节。就我们这儿的用途，只要知道这些就足够了：灵巧指针对象支持成员选择（**->**）和反引用（*****）这两个操作符，就象真的指针一样，并和内建指针一样是强类型的：你不能将一个指向 **T** 的灵巧指针指向一个非 **T** 类型的对象。

这儿是供引用计数对象使用的灵巧指针模板：

```
// template class for smart pointers-to-T objects. T must
// support the RCObject interface, typically by inheriting
// from RCObject
template<class T>
class RCPtr {
public:
  RCPtr(T* realPtr = 0);
  RCPtr(const RCPtr& rhs);
  ~RCPtr();
  RCPtr& operator=(const RCPtr& rhs);
  T* operator->() const;           // see Item 28
  T& operator*() const;            // see Item 28
private:
  T *pointee;                      // dumb pointer this
                                   // object is emulating
  void init();                     // common initialization
};
```

这个模板让灵巧指针对象控制在构造、赋值、析构时作什么操作。当这些事件发生时，这些对象可以自动地执行正确的操作来处理它们指向的对象的 **refCount** 字段。

例如，当一个 **RCPtr** 构建时，它指向的对象需要增加引用计数值。现在不需要程序员

手工处理这些细节了，因为 RCPtr 的构造函数自己处理它。两个构造函数几乎相同，除了初始化列表上的不同，为了不写两遍，我们将它放入一个名为 init 的私有成员函数中供二者调用：

```
template<class T>
RCPtr<T>::RCPtr(T* realPtr): pointee(realPtr)
{
  init();
}
template<class T>
RCPtr<T>::RCPtr(const RCPtr& rhs): pointee(rhs.pointee)
{
  init();
}
template<class T>
void RCPtr<T>::init()
{
  if (pointee == 0) {                // if the dumb pointer is
    return;                          // null, so is the smart one
  }
if (pointee->isShareable() == false) {          // if the value
    pointee = new T(*pointee);                  // isn't shareable,
  }                                             // copy it
pointee->addReference();             // note that there is now a
}                                    // new reference to the value
```

将相同的代码移入诸如 init 这样的一个独立函数是很值得效仿的，但它现在暗淡无光，因为在此处，这个函数的行为不正确。

问题是这个：当 init 需要创建 value 的一个新拷贝时（因为已存在的拷贝处于不可共享状态），它执行下面的代码：

```
pointee = new T(*pointee);
```

pointee 的类型是指向 T 的指针，所以这一语句构建了一个新的 T 对象，并用拷贝构造函数进行了初始化。由于 RCPtr 是在 String 类内部，T 将是 String::StringValue，所以上面的语句将调用 String::StringValue 的拷贝构造函数。我们没有为这个类申明拷贝构造函数，所以编译器将为我们生成一个。这个生成的拷贝构造函数遵守 C++的自动生成拷贝构造函数的原则，只拷贝了 StringValue 的数据 pointer，而没有拷贝所指向的 char *字符串。

这样的行为对几乎任何类（而不光是引用计数类）都是灾难，这就是为什么你应该养成为所有含有指针的类提供拷贝构造函数（和赋值运算）的习惯（见 **Item E11**）。

RCPtr<T>模板的正确行为取决于 **T** 含有正确的值拷贝行为（如深拷贝）的拷贝构造函数。我们必须在 **StringValue** 中增加这样的一个构造函数：

```
class String {
private:
  struct StringValue: public RCObject {
    StringValue(const StringValue& rhs);
    ...
  };
  ...
};
String::StringValue::StringValue(const StringValue& rhs)
{
  data = new char[strlen(rhs.data) + 1];
  strcpy(data, rhs.data);
}
```

深拷贝的构造函数的存在不是 **RCPtr<T>**的唯一假设。它还要求 **T** 从 **RCObject** 继承，或至少提供了 **RCObject** 的所提供的函数。事实上由于 **RCPtr** 对象只是被设计了指向引用计数对象的，这个假设并不过分。不过，这个假设必须被明确写入文档。

RCPtr<T>的最后一个假设是它所指向的对象类型为 **T**。这似乎是显然的。毕竟，**pointee** 的类型被申明为 **T***。但 **pointee** 可能实际上指向 **T** 的一个派生类。例如，如果我们有一个类 **SpecialStringValue** 是从 **String::StringValue** 继承的：

```
class String {
private:
  struct StringValue: public RCObject { ... };
  struct SpecialStringValue: public StringValue { ... };
  ...
};
```

我们可以生成一个 **String**，包容的 **RCPtr<StringValue>**指向一个 **SpecialStringValue** 对象。这时，我们希望 **init** 的这句：

```
pointee = new T(*pointee);                  // T is StringValue, but
                                            // pointee really points to
                                            // a SpecialStringValue
```

调用的是 SpecialStringValue 的拷贝构造函数，而不是 StringValue 的拷贝构造函数。我们可以提供使用虚拷贝构造函数（见 Item M25）来实现这一点。对于我们的 String 类，我们不期望从 StringValue 派生子类，所以我们忽略这个问题。

用这种方式实现了 RCPtr 的构造函数后，类的其它函数实现得很轻快。赋值运算很简洁明了，虽然“需要测试源对象的可共享状态”将问题稍微复杂化了。幸好，同样的问题已经在我们为构造函数写的 init 函数中处理了。我们可以爽快地再度使用它：

```
template<class T>
RCPtr<T>& RCPtr<T>::operator=(const RCPtr& rhs)
{
  if (pointee != rhs.pointee) {          // skip assignments
                                         // where the value
                                         // doesn't change
    if (pointee) {
      pointee->removeReference();        // remove reference to
    }                                    // current value
    pointee = rhs.pointee;               // point to new value
    init();                              // if possible, share it
  }                                      // else make own copy
  return *this;
}
```

析构函数很容易。当一个 RCPtr 被析构时，它只是简单地将它对引用计数对象的引用移除：

```
template<class T>
RCPtr<T>::~RCPtr()
{
  if (pointee)pointee->removeReference();
}
```

如果这个 RCPtr 是最后一个引用它的对象，这个对象将在 RCObject 的成员函数 removeReference 中被析构。因此，RCPtr 对象无需关心销毁它们指向的值的问题。

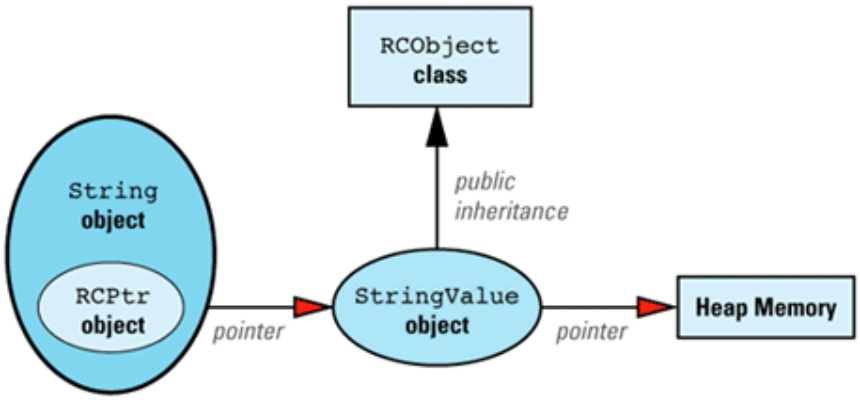
最后，RCPtr 的模拟指针的操作就是你在 Item M28 中看到的灵巧指针的部分：

```
template<class T>
T* RCPtr<T>::operator->() const { return pointee; }
template<class T>
T& RCPtr<T>::operator*() const { return *pointee; }
```

I 合在一起

够了!完结!最后,我们将各个部分放在一起,构造一个基于可重用的 **RCObject** 和 **RCPtr** 类的带引用计数的 **String** 类。或许，你还没有忘记这是我们的最初目标。

每个带引用计数的 **Sting** 对象被实现为这样的数据结构：



类的定义是：

```
template<class T>                    // template class for smart
class RCPtr {                        // pointers-to-T objects; T
public:                              // must inherit from RCObject
  RCPtr(T* realPtr = 0);
  RCPtr(const RCPtr& rhs);
  ~RCPtr();
  RCPtr& operator=(const RCPtr& rhs);
  T* operator->() const;
  T& operator*() const;
private:
  T *pointee;
  void init();
};
class RCObject {                     // base class for reference-
public:                              // counted objects
  void addReference();
  void removeReference();
  void markUnshareable();
```

```
  bool isShareable() const;
  bool isShared() const;
protected:
  RCObject();
  RCObject(const RCObject& rhs);
  RCObject& operator=(const RCObject& rhs);
  virtual ~RCObject() = 0;
private:
  int refCount;
  bool shareable;
};
class String {                          // class to be used by
public:                                 // application developers
  String(const char *value = "");
  const char& operator[](int index) const;
  char& operator[](int index);
private:
  // class representing string values
  struct StringValue: public RCObject {
    char *data;
    StringValue(const char *initValue);
    StringValue(const StringValue& rhs);
    void init(const char *initValue);
    ~StringValue();
  };
  RCPtr<StringValue> value;
};
```

绝大部分都是我们前面写的代码的翻新，没什么奇特之处。仔细检查后发现，我们在 **String::StringValue** 中增加了一个 **init** 函数，但，如我们下面将看到的，它的目的和 **RCPtr** 中的相同：消除构造函数中的重复代码。

这里有一个重大的不同：这个 **String** 类的公有接口和本条款开始处我们使用的版本不同。拷贝构造函数在哪里？赋值运算在哪里？析构函数在哪里？这儿明显有问题。

实际上，没问题。它工作得很好。如果你没看出为什么，需要重学 **C++**了（**prepare yourself for a C++ epiphany**）。

我们不再需要那些函数了！确实，**String** 对象的拷贝仍然被支持，并且，这个拷贝将正确处理藏在后面的被引用计数的 **StringValue** 对象，但 **String** 类不需要写下哪怕一行代码来让它发生。因为编译器为 **String** 自动生成的拷贝构造函数将自动调用其 **RCPtr** 成员的拷贝构造函数，而这个拷贝构造函数完成所有必须的对 **StringValue** 对象的操作，包括它的引用计数。**RCPtr** 是一个灵巧指针，所以这是它将完成的工作。它同样处理赋值和析构，所以 **String** 类同样不需要写出这些函数。我们的最初目的是将不可重用的引用计数代码从我们自己写的 **String** 类中移到一个与运行环境无关的类中以供任何其它类使用。现在，我们完成了这一点（用 **RCObject** 和 **RCPtr** 两个类），所以当它突然开始工作时别惊奇。它本来就应该能工作的。

将所以东西放在一起，这儿是 **RCObject** 的实现：

```
RCObject::RCObject()
: refCount(0), shareable(true) {}
RCObject::RCObject(const RCObject&)
: refCount(0), shareable(true) {}
RCObject& RCObject::operator=(const RCObject&)
{ return *this; }
RCObject::~RCObject() {}
void RCObject::addReference() { ++refCount; }
void RCObject::removeReference()
{ if (--refCount == 0) delete this; }
void RCObject::markUnshareable()
{ shareable = false; }
bool RCObject::isShareable() const
{ return shareable; }
bool RCObject::isShared() const
{ return refCount > 1; }
```

这是 **RCPtr** 的实现：

```
template<class T>
void RCPtr<T>::init()
{
  if (pointee == 0) return;
  if (pointee->isShareable() == false) {
    pointee = new T(*pointee);
  }
```

```
  pointee->addReference();
}
template<class T>
RCPtr<T>::RCPtr(T* realPtr)
: pointee(realPtr)
{ init(); }
template<class T>
RCPtr<T>::RCPtr(const RCPtr& rhs)
: pointee(rhs.pointee)
{ init(); }
template<class T>
RCPtr<T>::~RCPtr()
{ if (pointee)pointee->removeReference(); }
template<class T>
RCPtr<T>& RCPtr<T>::operator=(const RCPtr& rhs)
{
  if (pointee != rhs.pointee) {
    if (pointee) pointee->removeReference();
    pointee = rhs.pointee;
    init();
  }
return *this;
}
template<class T>
T* RCPtr<T>::operator->() const { return pointee; }
template<class T>
T& RCPtr<T>::operator*() const { return *pointee; }
```

这是 String::StringValue 的实现：

```
void String::StringValue::init(const char *initValue)
{
  data = new char[strlen(initValue) + 1];
  strcpy(data, initValue);
}
String::StringValue::StringValue(const char *initValue)
```

```
{ init(initValue); }
String::StringValue::StringValue(const StringValue& rhs)
{ init(rhs.data); }
String::StringValue::~StringValue()
{ delete [] data; }
```

最后，归结到 **String**，它的实现是：

```
String::String(const char *initValue)
: value(new StringValue(initValue)) {}
const char& String::operator[](int index) const
{ return value->data[index]; }
char& String::operator[](int index)
{
  if (value->isShared()) {
    value = new StringValue(value->data);
  }
  value->markUnshareable();
  return value->data[index];
}
```

如果你将它和我们用内建指针实现的版本相比较，你会受到两件事的打击。第一，代码有很多的减少。因为 **RCPtr** 完成了大量以前在 **String** 内部完成的处理引用计数的担子。第二，剩下的代码几乎没有变化：灵巧指针无缝替换了内建指针。实际上，唯一的变化是在 **operator[]**里，我们用调用 **isShared** 函数代替了直接检查 **refCount** 的值，并用灵巧指针 **RCPtr** 对象消除了写时拷贝时手工维护引用计数值的工作。

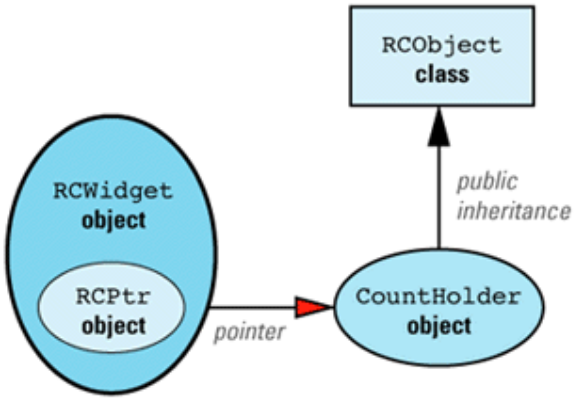
这当然全都很漂亮。谁能反对减少代码？谁能反对成功的封装？然而，这个全新的 **String** 类本身对用户的冲击远胜过它的实现细节，这才是真正的闪光点。如果没有什么消息是好消息的话，这本身就是最好的消息。**String** 的接口没有改变！我们增加了引用计数，我们增加了标记某个 **String** 的值为不可共享的能力，我们将引用计数功能移入一个新类，我们增加了灵巧指针来自动处理引用计数，但用户的一行代码都不需要修改。当然，我们改变了 **String** 类的定义，所以用户需要重新编译和链接，但他们在自己代码上的投资受到了完全的保护。你看到了吗？封装确实是个很好的东西。

■ 在现存类上增加引用计数

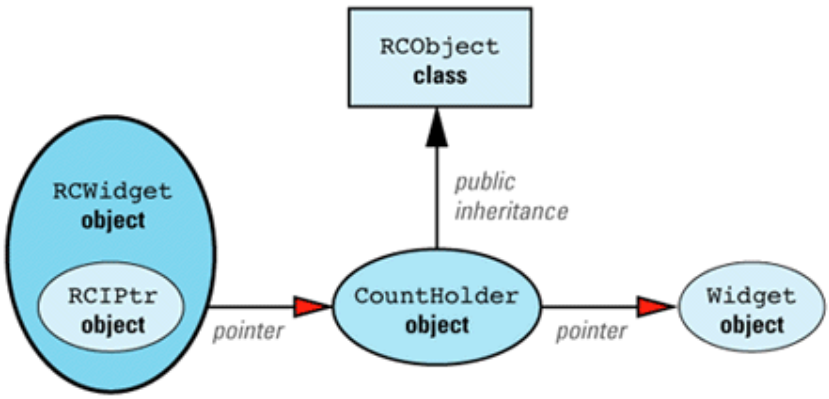
到现在为止，我们所讨论的都假设我们能够访问有关类的源码。但如果我们想让一个位于支撑库中而无法修改的类获得引用计数的好处呢？不可能让它们从 **RCObject** 继承的，所以也不能对它们使用灵巧指针 **RCPtr**。我们运气不好吗？

不是的。只要对我们的设计作小小的修改，我们就可以将引用计数加到任意类型上。

首先考虑如果从 **RCObject** 继承的话，我们的设计看起来将是什么样子。在这种情况下，我们需要增加一个类 **RCWidget** 以供用户使用，而所有的事情都和 **String/StringValue** 的例子一样，**RCWidget** 和 **String** 相同，**Widget** 和 **StringValue** 相同。设计看起来是这样的：



我们现在可以应用这句格言：计算机科学中的绝大部分问题都可以通过增加一个中间层次来解决。我们增加一个新类 **CountHolder** 以处理引用计数，它从 **RCObject** 继承。我们让 **CountHolder** 包含一个指针指向 **Widget**。然后用等价的灵巧指针 **RCIPter** 模板替代 **RCPtr** 模板，它知道 **CountHolder** 类的存在。（名字中的“**i**”表示间接“**indirect**”。）修改后的设计为：



如同 **StringValue** 一样，**CountHolder** 对用户而言，是 **RCWidget** 的实现细节。实际上，它是 **RCIPtr** 的实现细节，所以它嵌套在这个类中。**RCIPtr** 的实现如下：

```
template<class T>
```

```
class RCIPtr {
public:
  RCIPtr(T* realPtr = 0);
  RCIPtr(const RCIPtr& rhs);
  ~RCIPtr();
  RCIPtr& operator=(const RCIPtr& rhs);
  const T* operator->() const;              // see below for an
  T* operator->();                          // explanation of why
  const T& operator*() const;               // these functions are
  T& operator*();                           // declared this way
private:
  struct CountHolder: public RCObject {
    ~CountHolder() { delete pointee; }
    T *pointee;
  };
  CountHolder *counter;
  void init();
  void makeCopy();                                // see below
};
template<class T>
void RCIPtr<T>::init()
{
  if (counter->isShareable() == false) {
    T *oldValue = counter->pointee;
    counter = new CountHolder;
    counter->pointee = new T(*oldValue);
  }
  counter->addReference();
}
template<class T>
RCIPtr<T>::RCIPtr(T* realPtr)
: counter(new CountHolder)
{
  counter->pointee = realPtr;
```

```
  init();
}
template<class T>
RCIPtr<T>::RCIPtr(const RCIPtr& rhs)
: counter(rhs.counter)
{ init(); }
template<class T>
RCIPtr<T>::~RCIPtr()
{ counter->removeReference(); }
template<class T>
RCIPtr<T>& RCIPtr<T>::operator=(const RCIPtr& rhs)
{
  if (counter != rhs.counter) {
    counter->removeReference();
    counter = rhs.counter;
    init();
  }
  return *this;
}
template<class T>                          // implement the copy
void RCIPtr<T>::makeCopy()                 // part of copy-on-
{                                          // write (COW)
  if (counter->isShared()) {
    T *oldValue = counter->pointee;
    counter->removeReference();
    counter = new CountHolder;
    counter->pointee = new T(*oldValue);
    counter->addReference();
  }
}
template<class T>                           // const access;
const T* RCIPtr<T>::operator->() const      // no COW needed
{ return counter->pointee; }
template<class T>                           // non-const
```

```
T* RCIPtr<T>::operator->()                    // access; COW
{ makeCopy(); return counter->pointee; }      // needed
template<class T>                             // const access;
const T& RCIPtr<T>::operator*() const         // no COW needed
{ return *(counter->pointee); }
template<class T>                             // non-const
T& RCIPtr<T>::operator*()                     // access; do the
{ makeCopy(); return *(counter->pointee); } // COW thing
```

RCIPtr 与 RCPtr 只两处不同。第一，RCPtr 对象直接指向值对象，而 RCIptr 对象通过中间层的 CountHolder 对象指向值对象。第二，RCIPtr 重载了 operator->和 operator*，当有对被指向的对象的非 const 的操作时，写时拷贝自动被执行。

有了 RCIPtr，很容易实现 RCWidget，因为 RCWidget 的每个函数都是将调用传递给 RCIPtr 以操作 Widget 对象。举个例子，如果 Widget 是这样的：

```
class Widget {
public:
  Widget(int size);
  Widget(const Widget& rhs);
  ~Widget();
  Widget& operator=(const Widget& rhs);
  void doThis();
  int showThat() const;
};
```

那么 RCWidget 将被定义为这样：

```
class RCWidget {
public:
  RCWidget(int size): value(new Widget(size)) {}
  void doThis() { value->doThis(); }
  int showThat() const { return value->showThat(); }
private:
  RCIPtr<Widget> value;
};
```

注意 RCWidget 的构造函数是怎么用它被传入的参数调用 Widget 的构造函数的（通过 new 操作符，见 Item M8）；RCWidget 的 doThis 怎么调用 Widget 的 doThis 函数的；以及 RCWidget 的 showThat 怎么返回 Widget 的 showThat 的返回值的。同样要注意 RCWidget 没

有申明拷贝构造函数和赋值操作函数，也没有析构函数。如同 **String** 类一样，它不需要这些函数。感谢于 **RCIPtr** 的行为，**RCWidget** 的默认版本将完成正确的事情。

如果认为生成 **RCWidget** 的行为很机械，它应该自动进行，那么你是对的。不难写个小程序接受如 **Widget** 这样的类而输出 **RCWidget** 这样的类。如果你写了一个这样的程序，请让我知道。

I 评述

让我们从 **Widget**、**String**、值、灵巧指针和引用计数基类中摆脱一下。给个机会回顾一下，在更广阔的环境下看一下引用计数。在更大的环境下，我们必须处理一个更高层次的问题，也就是什么时候使用引用计数？

实现引用计数不是没有代价的。每个被引用的值带一个引用计数，其大部分操作都需要以某种形式检查或操作引用计数。对象的值需要更多的内存，而我们在处理它们时需要执行更多的代码。此外，就内部的源代码而言，带引用计数的类的复杂度比不带的版本高。没有引用计数的 **String** 类只依赖于自己，而我们最终的 **String** 类如果没有三个辅助类（**StringValue**、**RCObject** 和 **RCPtr**）就无法使用。确实，我们这个更复杂的设计确保在值可共享时的更高的效率；免除了跟踪对象所有权的需要，提高了引用计数的想法和实现的可重用性。但，这四个类必须写出来、被测试、文档化、和被维护，比单个类要多做更多的工作。即使是管理人员也能看出这点。

引用计数是基于对象通常共享相同的值的假设的优化技巧（参见 **Item M18**）。如果假设不成立的话，引用计数将比通常的方法使用更多的内存和执行更多的代码。另一方面，如果你的对象确实有具体相同值的趋势，那么引用计数将同时节省时间和空间。共享的值所占内存越大，同时共享的对象数目越多，节省的内存也就越大。创建和销毁这个值的代价越大，你节省的时间也越多。总之，引用计数在下列情况下对提高效率很有用：

少量的值被大量的对象共享。这样的共享通常通过调用赋值操作和拷贝构造而发生。对象/值的比例越高，越是适宜使用引用计数。

对象的值的创建和销毁代价很高昂，或它们占用大量的内存。即使这样，如果不是多个对象共享相同的值，引用计数仍然帮不了你任何东西。

只有一个方法来确认这些条件是否满足，而这个方法不是猜测或依赖直觉（见 **Item M16**）。这个方法是使用 **profiler** 或其它工具来分析。使用这种方法，你可以发现是否创建和销毁值的行为是性能瓶颈，并能得出对象/值的比例。只有当你手里有了这些数据，你才能得出是否从引用计数上得到的好处超过其缺点。

即使上面的条件满足了，使用引用计数仍然可能是不合适的。有些数据结构（如有向图）将导致自我引用或环状结构。这样的数据结构可能导致孤立的自引用对象，它没有被别人使用，而其引用计数又绝不会降到零。因为这个无用的结构中的每个对象被同结构中的至少一个对象所引用。商用化的垃圾收集体系使用特别的技术来查找这样的结构并消除它们，

但我们现在使用的这个简单的引用计数技术不是那么容易扩充出这个功能的。

即使效率不是主要问题，引用计数仍然很吸引人。如果你不放心谁应该去执行删除动作，那么引用计数正是这种让你放下担子的技巧。很多程序员只因为这个原因就使用引用计数。

让我们用最后一个问题结束讨论。当 **RCObject::removeReference** 减少对象的引用计数时，它检查新值是否为 **0**。如果是，**removeReference** 通过调用 **delete this** 销毁对象。这个操作只在对象是通过调用 **new** 生成时才安全，所以我们需要一些方法以确保 **RCObject** 只能用这种方法产生。

此处，我们用习惯方法来解决。**RCObject** 被设计为只作被引用计数的值对象的基类使用，而这些值对象应该只通过灵巧指针 **RCPtr** 引用。此外，值对象应该只能由值会共享的对象来实例化；它们不能被按通常的方法使用。在我们的例子中，值对象的类是 **StringValue**，我们通过将它申明为 **String** 的私有而限制其使用。只有 **String** 可以创建 **StringValue** 对象，所以 **String** 类的作者应该确保这些值对象都是通过 **new** 操作产成的。

于是，我们限制 **RCObject** 只能在堆上创建的方法就是指定一组满足这个要求的类，并确保只有这些类能创建 **RCObject** 对象。用户不可能无意地（或有意地）用一种不恰当的方法创建 **RCObject** 对象。我们限制了创建被引用计数对象的权力，当我们交出这个权力时，必须明确其附带条件是满足创建对象的限制条件。

l 注 10

标准 **C++**运行库中的 **string** 类型（见 **Item E49** 和 **Item M35**）同时使用了方法 **2** 和方法 **3**。从非 **const** 的 **operator[]**中返回的引用直到下一次的可能修改这个 **string** 的函数的调用为止都是有效的。在此之后，使用这个引用（或它指向的字符），其结果未定义。这样就它允许了：**string** 的可共享标志在调用可能修改 **string** 的函数时被重设为 **true**。

7.6 Item M30：代理类

虽然你和你的亲家可能住在同一地理位置，但就整个世界而言，通常不是这样的。很不幸，**C++**还没有认识到这个事实。至少，从它对数组的支持上可以看出一些迹象。在 **FORTRAN**、**BASIC** 甚至是 **COBOL** 中，你可以创二维、三维乃至 **n** 维数组（**OK**，**FORTRAN** 只能创最多 **7** 维数组，但别过于吹毛求疵吧）。但在 **C++**中呢？只是有时可以，而且也只是某种程度上的。

这是合法的：

```
int data[10][20];                          // 2D array: 10 by 20
```

而相同的结构如果使用变量作维的大小的话，是不可以的：

```
void processInput(int dim1, int dim2)
{
```

```
  int data[dim1][dim2];                    // error! array dimensions
  ...                                      // must be known during
}
```

甚至，在堆分配时都是不合法的：

```
int *data =
  new int[dim1][dim2];                     // error!
```

l 实现二维数组

多维数组在C++中的有用程度和其它语言相同,所以找到一个象样的支持方法是很重要的。常用方法是C++中的标准方法：用一个类来实现我们所需要的而C++语言中并没有提供的东西。因此，我们可以定义一个类模板来实现二维数组：

```
template<class T>
class Array2D {
public:
  Array2D(int dim1, int dim2);
  ...
};
```

现在，我们可以定义我们所需要的数组了：

```
Array2D<int> data(10, 20);             // fine
Array2D<float> *data =
  new Array2D<float>(10, 20);          // fine
void processInput(int dim1, int dim2)
{
  Array2D<int> data(dim1, dim2);       // fine
  ...
}
```

然而，使用这些array对象并不直接了当。根据C和C++中的语法习惯，我们应该能够使用[]来索引数组：

```
cout << data[3][6];
```

但我们在Array2D类中应该怎样申明下标操作以使得我们可以这么做?

我们最初的冲动可能是申明一个operator[][]函数：

```
template<class T>
class Array2D {
public:
  // declarations that won't compile
```

```
  T& operator[][](int index1, int index2);
  const T& operator[][](int index1, int index2) const;
  ...
};
```

然而，我们很快就会中止这种冲动，因为没有 **operator[][]**这种东西，别指望你的编译器能放过它。（所有可以重载的运算符见 **Item M7**。）我们得另起炉灶。

如果你能容忍奇怪的语法，你可能会学其它语言使用**()**来索引数组。这么做，你只需重载 **operator()**：

```
template<class T>
class Array2D {
public:
  // declarations that will compile
  T& operator()(int index1, int index2);
  const T& operator()(int index1, int index2) const;
  ...
};
```

用户于是这么使用数组：

```
cout << data(3, 6);
```

这很容易实现，并很容易推广到任意多维的数组。缺点是你的 **Array2D** 对象看起来和内嵌数组一点都不象。实际上，上面访问元素**(3,6)**的操作看起来相函数调用。

如果你拒绝让访问数组行为看起来象是从 **FORTRAN** 流窜过来的，你将再次会到使用**[]**上来。虽然没有 **operator[][]**，但写出下面这样的代码是合法的：

```
int data[10][20];
...
cout << data[3][6];          // fine
```

说明了什么？

说明，变量 **data** 不是真正的二维数组，它是一个 **10** 元素的一维数组。其中每一个元素又都是一个 **20** 元素的数组，所以表达式 **data[3][6]**实际上是**(data[3])[6]**，也就是 **data** 的第四个元素这个数组的第 **7** 个元素。简而言之，第一个**[]**返回的是一个数组，第二个**[]**从这个返回的数组中再去取一个元素。

我们可以通过重载 **Array2D** 类的 **operator[]**来玩同样的把戏。**Array2D** 的 **operator[]** 返回一个新类 **Array1D** 的对象。再重载 **Array1D** 的 **operator[]**来返回所需要的二维数组中的元素：

```
template<class T>
```

```
class Array2D {
public:
  class Array1D {
  public:
    T& operator[](int index);
    const T& operator[](int index) const;
    ...
  };
  Array1D operator[](int index);
  const Array1D operator[](int index) const;
  ...
};
```

现在，它合法了：

```
Array2D<float> data(10, 20);
...
cout << data[3][6];          // fine
```

这里，**data[3]**返回一个 **Array1d** 对象，在这个对象上的 **operator[]**操作返回二维数组中**(3,6)**位置上的浮点数。

Array2D 的用户并不需要知道 **Array1D** 类的存在。这个背后的“一维数组”对象从概念上来说，并不是为 **Array2D** 类的用户而存在的。其用户编程时就象他们在使用真正的二维数组一样。对于 **Array2D** 类的用户这样做是没有意义的：为了满足 **C++**的反复无常，这些对象必须在语法上兼容于其中的元素是另一个一维数组的一个一维数组。

每个 **Array1D** 对象扮演的是一个一维数组，而这个一维数组没有在使用 **Array2D** 的程序中出现。扮演其它对象的对象通常被称为代理类。在这个例子里，**Array1D** 是一个代理类。它的实例扮演的是一个在概念上不存在的一维数组。（术语代理对象（**proxy object**）和代理类（**proxy classs**）还不是很通用；这样的对象有时被叫做 **surrogate**。）

Ⅰ 区分通过 operator[]进行的是读操作还是写操作

使用代理来实现多维数组是很通用的的方法，但代理类的用途远不止这些。例如，**Item M5** 中展示了代理类可以怎样用来阻止单参数的构造函数被误用为类型转换函数。在代理类的各中用法中，最神奇的是帮助区分通过 **operator[]**进行的是读操作还是写操作。

考虑一下带引用计数而又支持 **operator[]**的 **string** 类型。这样的类的细节见于 **Item M29**。如果你还不了解引用记数背后的概念，那么现在就去熟悉 **Item M29** 中的内容将是个好主意。

支持 **operator[]**的 **string** 类型，允许用户些下这样的代码：

```
String s1, s2;          // a string-like class; the
                        // use of proxies keeps this
                        // class from conforming to
                        // the standard string
...                     // interface
cout << s1[5];          // read s1
s2[5] = 'x';            // write s2
s1[3] = s2[8];          // write s1, read s2
```

注意，**operator[]**可以在两种不同的情况下调用：读一个字符或写一个字符。读是个右值操作；写是个左值操作。（这个名词来自于编译器，左值出现在赋值运算的左边，右值出现在赋值运算的右边。）通常，将一个对象做左值使用意味着它可能被修改，做右值用意味着它不能够被修改。

我们想区分将 **operator[]**用作左值还是右值，因为，对于有引用计数的数据结构，读操作的代价可以远小于写操作的代价。如 **Item M29** 解释的，引用计数对象的写操作将导致整个数据结构的拷贝，而读不需要，只要简单地返回一个值。不幸的是，在 **operator[]**内部，没有办法确定它是怎么被调用的，不可能区分出它是做左值还是右值。

“但，等等，”你叫道，“我们不需要。我们可以基于 **const** 属性重载 **operator[]**，这样就可以区分读还是写了。”换句话说，你建议我们这么解决问题：

```
class String {
public:
  const char& operator[](int index) const;       // for reads
  char& operator[](int index);                    // for writes
  ...
};
```

唉，这不能工作。编译器根据调用成员函数的对象的 **const** 属性来选择此成员函数的 **const** 和非 **const** 版本，而不考虑调用时的环境。因此：

```
String s1, s2;
...
cout << s1[5];                      // calls non-const operator[],
                                    // because s1 isn't const
s2[5] = 'x';                        // also calls non-const
                                    // operator[]: s2 isn't const
s1[3] = s2[8];                      // both calls are to non-const
                                    // operator[], because both s1
```

```
                                        // and s2 are non-const objects
```

于是，重载 **operator[]**没能区分读还是写。

在 **Item M29** 中，我们屈从了这种不令人满意的状态，并保守地假设所有的 **operator[]** 调用都是写操作。这次，我们不会这么轻易放弃的。也许不可能在 **operator[]**内部区分左值还是右值操作，但我们仍然想区分它们。于是我们将去寻找一种方法。如果让你自己被其可能性所限制，生命还有什么快乐？

我们的方法基于这个事实：也许不可能在 **operator[]**内部区分左值还是右值操作，但我们仍然能区别对待读操作和写操作，如果我们将判断读还是写的行为推迟到我们知道 **operator[]**的结果被怎么使用之后的话。我们所需要的是有一个方法将读或写的判断推迟到 **operator[]**返回之后。（这是 **lazy** 原则（见 **Item M17**）的一个例子。）

proxy 类可以让我们得到我们所需要的时机，因为我们可以修改 **operator[]**让它返回一个（代理字符的）**proxy** 对象而不是字符本身。我们可以等着看这个 **proxy** 怎么被使用。如果是读它，我们可以断定 **operator[]**的调用是读。如果它被写，我们必须将 **operator[]** 的调用处理为写。

我们马上来看代码，但首先要理解我们使用的 **proxy** 类。在 **proxy** 类上只能做三件事：

- 创建它，也就是指定它扮演哪个字符。
- 将它作为赋值操作的目标，在这种情况下可以将赋值真正作用在它扮演的字符上。这样被使用时，**proxy** 类扮演的是左值。
- 用其它方式使用它。这时，代理类扮演的是右值。

这里是一个被带引用计数的 **string** 类用作 **proxy** 类以区分 **operator[]**是作左值还是右值使用的例子：

```
class String {                      // reference-counted strings;
public:                             // see Item 29 for details
  class CharProxy {                 // proxies for string chars
  public:
    CharProxy(String& str, int index);              // creation
    CharProxy& operator=(const CharProxy& rhs);     // lvalue
    CharProxy& operator=(char c);                   // uses
    operator char() const;                          // rvalue
                                                    // use
  private:
    String& theString;              // string this proxy pertains to
    int charIndex;                  // char within that string
                                    // this proxy stands for
```

```
  };
  // continuation of String class
  const CharProxy
    operator[](int index) const;   // for const Strings
  CharProxy operator[](int index); // for non-const Strings
  ...
friend class CharProxy;
private:
  RCPtr<StringValue> value;
};
```

除了增加的 **CharProxy** 类（我们将在下面讲解）外，这个 **String** 类与 **Item M29** 中的最终版本相比，唯一不同之处就是所有的 **operator[]**函数现在返回的是 **CharProxy** 对象。然而，**String** 类的用户可以忽略这一点，并当作 **operator[]**返回的仍然是通常形式的字符（或其引用，见 **Item M1**）来编程：

```
String s1, s2;          // reference-counted strings
                        // using proxies
...
cout << s1[5];          // still legal, still works
s2[5] = 'x';            // also legal, also works
s1[3] = s2[8];          // of course it's legal,
                        // of course it works
```

有意思的不是它能工作，而是它为什么能工作。

先看这条语句：

```
cout << s1[5];
```

表达式 **s1[5]**返回的是一 **CharProxy** 对象。没有为这样的对象定义输出流操作，所以编译器努力地寻找一个隐式的类型转换以使得 **operator<<**调用成功（见 **Item M5**）。它们找到一个：在 **CahrProxy** 类内部申明了一个隐式转换到 **char** 的操作。于是自动调用这个转换操作，结果就是 **CharProxy** 类扮演的字符被打印输出了。这个 **CharProxy** 到 **char** 的转换是所有代理对象作右值使用时发生的典型行为。

作左值时的处理就不一样了。再看：

```
s2[5] = 'x';
```

和前面一样，表达式 **s2[5]**返回的是一个 **CharProxy** 对象，但这次它是赋值操作的目标。由于赋值的目标是 **CharProxy** 类，所以调用的是 **CharProxy** 类中的赋值操作。这至关重要，因为在 **CharProxy** 的赋值操作中，我们知道被赋值的 **CharProxy** 对象是作左值使用的。因此，

我们知道 proxy 类扮演的字符是作左值使用的，必须执行一些必要的操作以实现字符的左值操作。

同理，语句

```
s1[3] = s2[8];
```

调用作用于两个 CharProxy 对象间的赋值操作，在此操作内部，我们知道左边一个是作左值，右边一个作右值。

“呀，呀，呀！”你叫道，“快给我看。”OK，这是 String 的 opertator[]函数的代码：

```
const String::CharProxy String::operator[](int index) const
{
  return CharProxy(const_cast<String&>(*this), index);
}
String::CharProxy String::operator[](int index)
{
  return CharProxy(*this, index);
}
```

每个函数都创建和返回一个 proxy 对象来代替字符。根本没有对那个字符作任何操作：我们将它推迟到直到我们知道是读操作还是写操作。

注意，operator[]的 const 版本返回一个 const 的 proxy 对象。因为 CharProxy::operator=是个非 const 的成员函数，这样的 proxy 对象不能作赋值的目标使用。因此，不管是从 operator[]的 const 版本返回的 proxy 对象，还是它所扮演的字符都不能作左值使用。很方便啊，它正好是我们想要的 const 版本的 operator[]的行为。

同样要注意在 const 的 operator[]返回而创建 CharProxy 对象时，对*this 使用的 const_cast（见 Item M2）。这使得它满足了 CharProxy 类的构造函数的需要，它的构造函数只接受一个非 const 的 String 类。类型转换通常是领人不安的，但在此处，operator[]返回的 CharProxy 对象自己是 const 的，所以不用担心 String 内部的字符可能被通过 proxy 类被修改。

通过 operator[]返回的 proxy 对象记录了它属于哪个 string 对象以及所扮演的字符的下标：

```
String::CharProxy::CharProxy(String& str, int index)
: theString(str), charIndex(index) {}
```

将 proxy 对象作右值使用时很简单——只需返回它所扮演的字符就可以了：

```
String::CharProxy::operator char() const
{
  return theString.value->data[charIndex];
```

```
}
```

如果已经忘了 **string** 对象的 **value** 成员和它指向的 **data** 成员的关系的话，请回顾一下 **Item M29** 以增强记忆。因为这个函数返回了一个字符的值，并且又因为 **C++**限定这样通过值返回的对象只能作右值使用，所以这个转换函数只能出现在右值的位置。

回头再看 **CahrProxy** 的赋值操作的实现，这是我们必须处理 **proxy** 对象所扮演的字符作赋值的目标（即左值）使用的地方。我们可以将 **CharProxy** 的赋值操作实现如下：

```
String::CharProxy&
String::CharProxy::operator=(const CharProxy& rhs)
{
  // if the string is sharing a value with other String objects,
  // break off a separate copy of the value for this string only
  if (theString.value->isShared()) {
    theString.value = new StringValue(theString.value->data);
  }
  // now make the assignment: assign the value of the char
  // represented by rhs to the char represented by *this
  theString.value->data[charIndex] =
    rhs.theString.value->data[rhs.charIndex];
  return *this;
}
```

如果与 **Item M29** 中的非 **const** 的 **String::operator[]**进行比较，你将看到它们极其相似。这是预料之中的。在 **Item M29** 中，我们悲观地假设所有非 **const** 的 **operator[]**的调用都是写操作，所以实现成这样。现在，我们将写操作的实现移入 **CharProxy** 的赋值操作中，于是可以避免非 **const** 的 **operator[]**的调用只是作右值时所多付出的写操作的代价。随便提一句，这个函数需要访问 **string** 的私有数据成员 **value**。这是前面将 **CharProxy** 申明为 **string** 的友元的原因。

第二个 **CharProxy** 的赋值操作是类似的：

```
String::CharProxy& String::CharProxy::operator=(char c)
{
  if (theString.value->isShared()) {
    theString.value = new StringValue(theString.value->data);
  }
  theString.value->data[charIndex] = c;
  return *this;
```

```
}
```

作为一个资深的软件工程师， 你当然应该消除这两个赋值操作中的代码重复，应该将它们放入一个私有成员函数中供二者调用。你不是这样的人吗？

l 局限性

使用 **proxy** 类是个区分 **operator[]**作左值还是右值的好方法，但它不是没有缺点的。我们很喜欢 **proxy** 对象对其所扮演的对象的无缝替代，但这很难实现。这是因为，右值不只是出现在赋值运算的情况下，那时，**proxy** 对象的行为就和实际的对象不一致了。

再看一下来自于 **Item M29** 的代码，以证明我们为什么为每个 **StringValue** 对象增加一个共享标志。如果 **String::operator[]**返回一个 **CharProxy** 而不是 **char &**，下面的代码将不能编译：

```
String s1 = "Hello";
char *p = &s1[1];              // error!
```

表达式 **s1[1]**返回一个 **CharProxy**，于是“=”的右边是一个 **CharProxy ***。没有从 **CharProxy ***到 **char ***的转换函数，所以 **p** 的初始化过程编译失败了。通常，取 **proxy** 对象地址的操作与取实际对象地址的操作得到的指针，其类型是不同的。

要消除这个不同，你需要重载 **CharProxy** 类的取地址运算：

```
class String {
public:
  class CharProxy {
  public:
    ...
    char * operator&();
    const char * operator&() const;
    ...
  };
  ...
};
```

这些函数很容易实现。**const** 版本返回其扮演的字符的 **const** 型的指针：

```
const char * String::CharProxy::operator&() const
{
  return &(theString.value->data[charIndex]);
}
```

非 **const** 版本有多一些操作，因为它返回的指针指项的字符可以被修改。它和 **Item M29** 中的非 **const** 的 **String::operator[]**行为相似，实现也很接近：

```
char * String::CharProxy::operator&()
{
  // make sure the character to which this function returns
  // a pointer isn't shared by any other String objects
  if (theString.value->isShared()) {
    theString.value = new StringValue(theString.value->data);
  }
  // we don't know how long the pointer this function
  // returns will be kept by clients, so the StringValue
  // object can never be shared
  theString.value->markUnshareable();
  return &(theString.value->data[charIndex]);
}
```

其代码和 **CharProxy** 的其它成员函数有很多相同，所以我知道你将把它们封入一个私有成员函数。

char 和代理它的 **CharProxy** 的第二个不同之处出现带引用计数的数组模板中如果我们想用 **proxy** 类来区分其 **operator[]**作左值还是右值时：

```
template<class T>                       // reference-counted array
class Array {                           // using proxies
public:
  class Proxy {
  public:
    Proxy(Array<T>& array, int index);
    Proxy& operator=(const T& rhs);
    operator T() const;
    ...
  };
  const Proxy operator[](int index) const;
  Proxy operator[](int index);
  ...
};
```

看一下这个数组可能被怎样使用：

```
Array<int> intArray;
...
```

```
intArray[5] = 22;                    // fine
intArray[5] += 5;                    // error!
++intArray[5];                       // error!
```

如我们所料，当 operator[]作最简单的赋值操作的目标时，是成功的，但当它出现 operator+=和 operator++的左侧时，失败了。因为 operator[]返回一个 proxy 对象，而它没有 operator+=和 operator++操作。同样的情况存在于其它需要左值的操作中，包括 operator*=、operator<<=、operator--等等。如果你想让这些操作你作用在 operator[]上，必须为 Arrar<T>::Proxy 类定义所有这些函数。这是一个极大量的工作，你可能不愿意去做的。不幸的是，你要么去做这些工作，要么没有这些操作，不能两全。

一个类似的问题必须面对：通过 proxy 对象调用实际对象的成员函数。想避开它是不可能的。例如，假设我们用带引用计数的数组处理有理数。我们将定义一个 Rational 类，然后使用前面看到的 Array 模板：

```
class Rational {
public:
  Rational(int numerator = 0, int denominator = 1);
  int numerator() const;
  int denominator() const;
  ...
};
Array<Rational> array;
```

这是我们所期望的使用方式，但我们很失望：

```
cout << array[4].numerator();                    // error!
int denom = array[22].denominator();             // error!
```

现在，不同之处很清楚了；operator[]返回一个 proxy 对象而不是实际的 Rational 对象。但成员函数 numerator()和 denominator()只存在于 Rational 对象上，而不是其 proxy 对象。因此，你的编译器发出了抱怨。要使得 proxy 对象的行为和它们所扮演的对象一致，你必须重载可作用于实际对象的每一个函数。

另一个 proxy 对象替代实际对象失败的情况是作为非 const 的引用传给函数：

```
void swap(char& a, char& b);                    // swaps the value of a and
b
String s = "+C+";                                // oops, should be "C++"
swap(s[0], s[1]);                                // this should fix the
                                                 // problem, but it won't
                                                 // compile
```

String::operator[]返回一个 CharProxy 对象，但 swap()函数要求它所参数是 char & 类型。一个 CharProxy 对象可以印式地转换为一个 char，但没有转换为 char &的转换函数。而它可能转换成的 char 并不能成为 swap 的 char &参数，因为这个 char 是一个临时对象（它是 operator char()的返回值），根据 Item M19 的解释，拒绝将临时对象绑定为非 const 的引用的形参是有道理的。

最后一种 proxy 对象不能无缝替换实际对象的情况是隐式类型转换。当 proxy 对象隐式转换为它所扮演的实际对象时，一个用户自定义的转换函数被调用了。例如，一个 CharProxy 对象可以转换为它扮演的 char，通过调用 operator char()函数。如 Item M5 解释的，编译器在调用函数而将参数转换为此函数所要的类型时，只调用一个用户自定义的转换函数。于是，很可能在函数调用时，传实际对象是成功的而传 proxy 对象是失败的。例如，我们有一个 TVStation 类和一个函数 watchTV()：

```
class TVStation {
public:
  TVStation(int channel);
  ...
};
void watchTV(const TVStation& station, float hoursToWatch);
```

借助于 int 到 TVStation 的隐式类型转换（见 Item M5），我们可以这么做：

```
watchTV(10, 2.5);                    // watch channel 10 for
                                     // 2.5 hours
```

然而，当使用那个用 proxy 类区分 operator[]作左右值的带引用计数的数组模板时，我们就不能这么做了：

```
Array<int> intArray;
intArray[4] = 10;
watchTV(intArray[4], 2.5);           // error! no conversion
                                     // from Proxy<int> to
                                     // TVStation
```

由于问题发生在隐式类型转换上，它很难解决。实际上，更好的设计应该是申明它的构造函数为 explicit，以使得第一次的调用 watchTV()的行为都编译失败。关于隐式类型转换的细节和 explicit 对此的影响见 Item M5。

I 评述

Proxy 类可以完成一些其它方法很难甚至不可能实现的行为。多维数组是一个例子，左/右值的区分是第二个，限制隐式类型转换（见 Item M5）是第三个。

同时，proxy 类也有缺点。作为函数返回值，proxy 对象是临时对象（见 Item 19），它

们必须被构造和析构。这不是免费的，虽然此付出能从具备了区分读写的能力上得到更多的补偿。**Proxy** 对象的存在增加了软件的复杂度，因为额外增加的类使得事情更难设计、实现、理解和维护。

最后，从一个处理实际对象的类改换到处理 **proxy** 对象的类经常改变了类的语义，因为 **proxy** 对象通常表现出的行为与实际对象有些微妙的区别。有时，这使得在设计系统时无法选择使用 **proxy** 对象，但很多情况下很少有操作需要将 **proxy** 对象暴露给用户。例如，很少有用户取上面的二维数组例子中的 **Array1D** 对象的地址，也不怎么有可能将下标索引的对象（见 **Item M5**）作参数传给一个期望其它类型的函数。在很多情况下，**proxy** 对象可以完美替代实际对象。当它们可以工作时，通常也是没有其它方法可采用的情况。

7.7 Item M31：让函数根据一个以上的对象来决定怎么虚拟

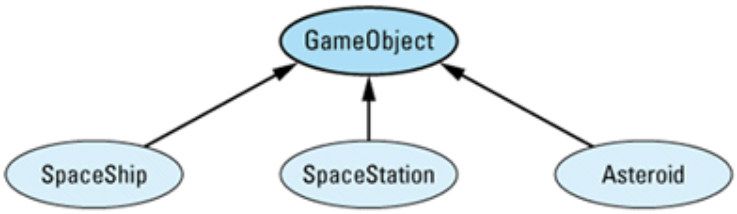
有时，借用一下 **Jacqueline Susann** 的话：一次是不够的。例如你有着一个光辉形象、崇高声望、丰厚薪水的程序员工作，在 **Redmond，Wshington** 的一个著名软件公司——当然，我说的就是任天堂。为了得到经理的注意，你可能决定编写一个 **video game**。游戏的背景是发生在太空，有宇宙飞船、太空站和小行星。

在你构造的世界中的宇宙飞船、太空站和小行星，它们可能会互相碰撞。假设其规则是：

- 如果飞船和空间站以低速接触，飞船将泊入空间站。否则，它们将有正比于相对速度的损坏。
- 如果飞船与飞船，或空间站与空间站相互碰撞，参与者均有正比于相对速度的损坏。
- 如果小行星与飞船或空间站碰撞，小行星毁灭。如果是小行星体积较大，飞船或空间站也毁坏。
- 如果两个小行星碰撞，将碎裂为更小的小行星，并向各个方向溅射。

这好象是个无聊的游戏，但用作我们的例子已经足够了，考虑一下怎么组织 **C++**代码以处理物体间的碰撞。

我们从分析飞船、太空站和小行星的共同特性开始。至少，它们都在运动，所以有一个速度来描述这个运动。基于这一点，自然而然地设计一个基类，而它们可以从此继承。实际上，这样的类几乎总是抽象基类，并且，如果你留心我在 **Item M33** 中的警告，基类总是抽象的。所以，继承体系是这样的：




```
class GameObject { ... };
class SpaceShip: public GameObject { ... };
class SpaceStation: public GameObject { ... };
class Asteroid: public GameObject { ... };
```

现在，假设你开始进入程序内部，写代码来检测和处理物体间的碰撞。你会提出这样一个函数：

```
void checkForCollision(GameObject& object1,
                       GameObject& object2)
{
  if (theyJustCollided(object1, object2)) {
    processCollision(object1, object2);
  }
  else {
    ...
  }
}
```

问题来了。当你调用 **processCollision()**时，你知道 **object1** 和 **object2** 正好相撞，并且你知道发生的结果将取决于**object1**和**object2**的真实类型,但你并不知道其真实类型；你所知道的就只有它们是 **GameObject** 对象。如果碰撞的处理过程只取决于 **object1** 的动态类型，你可以将 **processCollision()** 设为虚函数，并调用 **object1.processColliion(object2)**。如果只取决于 **object2** 的动态类型,也可以同样处理。但现在，取决于两个对象的动态类型。虚函数体系只能作用在一个对象身上，它不足以解决问题。

你需要的是一种作用在多个对象上的虚函数。**C++**没有提供这样的函数。可是，你必须要实现上面的要求。现在怎么办呢？

一种办法是扔掉 **C++**，换种其它语言。比如，你可以改用 **CLOS**（**Common Lisp Object System**）。**CLOS** 支持绝大部分面向对象的函数调用体系中只能想象的东西：**multi-method**。**multi-method** 是在任意多的参数上虚拟的函数，并且 **CLOS** 更进一步的提供了明确控制“被

重载的 multi-method 将如何调用”的特性。

让我们假设，你必须用 C++实现，所以必须找到一个方法来解决这个被称为“二重调度（double dispatch）”的问题。（这个名字来自于 object-oriented programming community，在那里虚函数调用的术语是“message dispatch”，而基两个参数的虚调用是通过“double dispatch”实现的，推而广之，在多个参数上的虚函数叫“multiple dispatch”。）有几个方法可以考虑。但没有哪个是没有缺点的，这不该奇怪。C++没有直接提供“double dispatch”，所以你必须自己完成编译器在实现虚函数时所做的工作（见 Item M24）。如果容易的话，我们可能就什么都自己做了，并用 C 语言编程了。我们没有，而且我们也不能够，所以系紧你的安全带，有一个坎途了。

I 用虚函数加 RTTI

虚函数实现了一个单一调度，这只是我们所需要的一半；编译器为我们实现虚函数，所以我们在 GameObject 中申明一个虚函数 collide。这个函数被派生类以通常的形式重载：

```
class GameObject {
public:
  virtual void collide(GameObject& otherObject) = 0;
  ...
};
class SpaceShip: public GameObject {
public:
  virtual void collide(GameObject& otherObject);
  ...
};
```

我在这里只写了派生类 SpaceShip 的情况，SpaceStation 和 Asteroid 的形式完全一样的。

实现二重调度的最常见方法就是和虚函数体系格格不入的 if...then...else 链。在这种刺眼的体系下，我们首先是发现 otherObject 的真实类型，然后测试所有的可能：

```
// if we collide with an object of unknown type, we
// throw an exception of this type:
class CollisionWithUnknownObject {
public:
  CollisionWithUnknownObject(GameObject& whatWeHit);
  ...
};
void SpaceShip::collide(GameObject& otherObject)
```

```
{
  const type_info& objectType = typeid(otherObject);
  if (objectType == typeid(SpaceShip)) {
    SpaceShip& ss = static_cast<SpaceShip&>(otherObject);
    process a SpaceShip-SpaceShip collision;
  }
  else if (objectType == typeid(SpaceStation)) {
    SpaceStation& ss =
      static_cast<SpaceStation&>(otherObject);
    process a SpaceShip-SpaceStation collision;
  }
  else if (objectType == typeid(Asteroid)) {
    Asteroid& a = static_cast<Asteroid&>(otherObject);
    process a SpaceShip-Asteroid collision;
  }
  else {
    throw CollisionWithUnknownObject(otherObject);
  }
}
```

注意，我们需要检测的只是一个对象的类型。另一个是***this**，它的类型由虚函数体系判断。我们现在处于 **SpaceShip** 的成员函数中，所以***this** 肯定是一个 **SpaceShip** 对象，因此我们只需找出 **otherObject** 的类型。

这儿的代码一点都不复杂。它很容易实现。也很容易让它工作。**RTTI** 只有一点令人不安：它只是看起来无害。实际的危险来自于最后一个 **else** 语句，在这儿抛了一个异常。

我们的代价是几乎放弃了封装，因为每个 **collide** 函数都必须知道所以其它同胞类中的版本。尤其是，如果增加一个新的类时，我们必须更新每一个基于 **RTTI** 的 **if...then...else** 链以处理这个新的类型。即使只是忘了一处，程序都将有一个 **bug**，而且它还不显眼。编译器也没办法帮助我们检查这种疏忽，因为它们根本不知道我们在做什么（参见 **Item E39**）。

这种类型相关的程序在 **C** 语言中已经很有一段历史了，而我们也知道，这样的程序本质上是没有可维护性的。扩充这样的程序最终是不可想象的。这是引入虚函数的主意原因：将产生和维护类型相关的函数调用的担子由程序员转给编译器。当我们用 **RTTI** 实现二重调度时，我们正退回到过去的苦日子中。

这种过时的技巧在 **C** 语言中导致了错误，它们 **C++**语言也仍然导致错误。认识到我们自

己的脆弱，我们在 **collide** 函数中加上了最后的那个 **else** 语句，以处理如果遇到一个未知类型。这种情况原则上说是不可能发生的，但在我们决定使用 **RTTI** 时又怎么知道呢？有很多种方法来处理这种未曾预料的相互作用，但没有一个令人非常满意。在这个例子里，我们选择了抛出一个异常，但无法想象调用者对这个错误的处理能够比我们好多少，因为我们遇到了一个我们不知道其存在的东西。

Ⅰ 只使用虚函数

其实有一个方法可以将用 **RTTI** 实现二重调度固有风险降到最低的，不过在此之前让我们看一下怎么只用虚函数来解决二重调度问题。这个方法和 **RTTI** 方法有这同样的基本构架。**collide** 函数被申明为虚，并被所有派生类重定义，此外，它还被每个类重载，每个重载处理一个派生类型：

```
class SpaceShip;                          // forward declarations
class SpaceStation;
class Asteroid;
class GameObject {
public:
  virtual void collide(GameObject&      otherObject) = 0;
  virtual void collide(SpaceShip&       otherObject) = 0;
  virtual void collide(SpaceStation&    otherObject) = 0;
  virtual void collide(Asteroid&        otherobject) = 0;
  ...
};
class SpaceShip: public GameObject {
public:
  virtual void collide(GameObject&       otherObject);
  virtual void collide(SpaceShip&        otherObject);
  virtual void collide(SpaceStation&     otherObject);
  virtual void collide(Asteroid&         otherobject);
  ...
};
```

其基本原理就是用两个单一调度实现二重调度，也就是说有两个单独的虚函数调用：第一次决定第一个对象的动态类型，第二次决定第二个对象动态类型。和前面一样，第一次虚函数调用带的是 **GameObject** 类型的参数。其实现是令人吃惊地简单：

```
void SpaceShip::collide(GameObject& otherObject)
{
```

```
  otherObject.collide(*this);
}
```

粗一看，它象依据参数的顺序进行循环调用，也就是开始的 **otherObject** 变成了调用成员函数的对象，而***this** 成了它的参数。但再仔细看一下啦，它不是循环调用。你知道的，编译器根据参数的静态类型决定调那一组函数中的哪一个。在这儿，有四个不同的 **collide** 函数可以被调用，但根据***this** 的静态类型来选中其中一个。现在的静态类型是什么？因为是在 **SpaceShip** 的成员函数中，所以***this** 肯定是 **SpaceShip** 类型。调用的将是接受 **SpaceShip** 参数的 **collide** 函数，而不是带 **GameOjbect** 类型参数的 **collide** 函数。

所有的 **collide** 函数都是虚函数，所以在 **SpaceShip::collide** 中调用的是 **otherObject** 真实类型中实现的 **collide** 版本。在这个版本中，两个对象的真实类型都是知道的，左边的是***this**（实现这个函数的类的类型），右边对象的真实类型是 **SpaceShip**（申明的形参类型）。

看了 **SpaceShip** 类中的其它 **collide** 的实现，就更清楚了：

```
void SpaceShip::collide(SpaceShip& otherObject)
{
  process a SpaceShip-SpaceShip collision;
}
void SpaceShip::collide(SpaceStation& otherObject)
{
  process a SpaceShip-SpaceStation collision;
}
void SpaceShip::collide(Asteroid& otherObject)
{
  process a SpaceShip-Asteroid collision;
}
```

你看到了，一点都不混乱，也不麻烦，没有 **RTTI**，也不需要为意料之外的对象类型抛异常。不会有意料之外的类型的，这就是使用虚函数的好处。实际上，如果没有那个致命缺陷的话，它就是实现二重调度问题的完美解决方案。

这个缺陷是，和前面看到的 **RTTI** 方法一样：每个类都必须知道它的同胞类。当增加新类时，所有的代码都必须更新。不过，更新方法和前面不一样。确实，没有 **if...then...else** 需要修改，但通常是更差：每个类都需要增加一个新的虚函数。就本例而言，如果你决定增加一个新类 **Satellite**（继承于 **GameObjcet**），你必须为每个现存类增加一个 **collide** 函数。

修改现存类经常是你做不到的。比如，你不是在写整个游戏，只是在完成程序框架下的一个支撑库，你可能无权修改 **GameObject** 类或从其经常的框架类。此时，增加一个新的成员函数（虚的或不虚的），都是不可能的。也就说，你理论上有操作需要被修改的类的权

限，但实际上没有。打个比方，你受雇于 Nitendo，使用一个包含 GameObject 和其它需要的类的运行库进行编程。当然不是只有你一个人在使用这个库，全公司都将震动于每次你决定在你的代码中增加一个新类型时，所有的程序都需要重新编译。实际中，广被使用的库极少被修改，因为重新编译所有用了这个库的程序的代价太大了。（参见 Item M34，以了解怎么设计将编译依赖度降到最低的运行库。）

总结一下就是：如果你需要实现二重调度，最好的办法是修改设计以取消这个需要。如果做不到的话，虚函数的方法比 RTTI 的方法安全，但它限制了你的程序的可控制性（取决于你是否有权修改头文件）。另一方面，RTTI 的方法不需要重编译，但通常会导致代码无法维护。自己做抉择啦！

I 模拟虚函数表

有一个方法来增加选择。你可以回顾 Item M24，编译器通常创建一个函数指针数组（vtbl）来实现虚函数，并在虚函数被调用时在这个数组中进行下标索引。使用 vtbl，编译器避免了使用 if...then...else 链，并能在所有调用虚函数的地方生成同样的代码：确定正确的 vtbl 下标，然后调用 vtbl 这个位置上存储的指针所指向的函数。

没理由说你不能这么做。如果这么做了，不但使得你基于 RTTI 的代码更具效率（下标索引加函数指针的反引用通常比 if...then...else 高效，产生的代码也少），同样也将 RTTI 的使用范围限定在一处：你初始化函数指针数组的地方。提醒一下，看下面的内容前最好做一下深呼吸（ I should mention that the meek may inherit the earth, but the meek of heart may wish to take a few deep breaths before reading what follows）。

对 GameObjcet 继承体系中的函数作一些修改：

```
class GameObject {
public:
  virtual void collide(GameObject& otherObject) = 0;
  ...
};
class SpaceShip: public GameObject {
public:
  virtual void collide(GameObject& otherObject);
  virtual void hitSpaceShip(SpaceShip& otherObject);
  virtual void hitSpaceStation(SpaceStation& otherObject);
  virtual void hitAsteroid(Asteroid& otherobject);
  ...
};
void SpaceShip::hitSpaceShip(SpaceShip& otherObject)
```

```
{
  process a SpaceShip-SpaceShip collision;
}
void SpaceShip::hitSpaceStation(SpaceStation& otherObject)
{
  process a SpaceShip-SpaceStation collision;
}
void SpaceShip::hitAsteroid(Asteroid& otherObject)
{
  process a SpaceShip-Asteroid collision;
}
```

和开始时使用的基于 **RTTI** 的方法相似，**GameObjcet** 类只有一个处理碰撞的函数，它实现必须的二重调度的第一重。和后来的基于虚函数的方法相似，每种碰撞都由一个独立的函数处理，不过不同的是，这次，这些函数有着不同的名字，而不是都叫 **collide**。放弃重载是有原因的，你很快就要见到的。注意，上面的设计中，有了所有其它需要的东西，除了没有实现 **Spaceship::collide**（这是不同的碰撞函数被调用的地方）。和以前一样，实现了 **SpaceShip** 类，**SpaceStation** 类和 **Asteroid** 类也就出来了。

在 **SpaceShip::collide** 中，我们需要一个方法来映射参数 **otherObject** 的动态类型到一个成员函数指针（指向一个适当的碰撞处理函数）。一个简单的方法是创建一个映射表，给定的类名对应恰当的成员函数指针。直接使用一个这样的映射表来实现 **collide** 是可行的，但如果增加一个中间函数 **lookup** 时，将更好理解。**lookup** 函数接受一个 **GameObject** 参数，返回相应的成员函数指针。

这是 **lookup** 的申明：

```
class SpaceShip: public GameObject {
private:
  typedef void (SpaceShip::*HitFunctionPtr)(GameObject&);
  static HitFunctionPtr lookup(const GameObject& whatWeHit);
  ...
};
```

函数指针的语法不怎么优美，而成员函数指针就更差了，所以我们作了一个类型重定义。

既然有了 **lookup**，**collide** 的实现如下：

```
void SpaceShip::collide(GameObject& otherObject)
{
```

```
  HitFunctionPtr hfp =
    lookup(otherObject);              // find the function to call
  if (hfp) {                          // if a function was found
    (this->*hfp)(otherObject);        // call it
  }
  else {
    throw CollisionWithUnknownObject(otherObject);
  }
}
```

如果我们能保持映射表和 **GameObject** 的继承层次的同步，**lookup** 就总能找到传入对象对应的有效函数指针。人终究只是人，就算再仔细，错误也会钻入软件。这就是我们为什么检查 **lookup** 的返回值并在其失败时抛异常的原因。

剩下的就是实现 **lookup** 了。提供了一个对象类型到成员函数指针的映射表后，**lookup** 自己很容易实现，但创建、初始化和析构这个映射表是个有意思的问题。

这样的数组应该在它被使用前构造和初始化，并在不再被需要时析构。我们可以使用 **new** 和 **delete** 来手工创建和析构它，但这时怎么保证在初始化以前不被使用呢？更好的解决方案是让编译器自动完成，在 **lookup** 中把这个数组申明为静态就可以了。这样，它在第一次调用 **lookup** 前构造和初始化，在 **main** 退出后的某个时刻被自动析构（见 **Item E47**）。

而且，我们可以使用标准模板库提供的 **map** 模板来实现映射表，因为这正是 **map** 的功能：

```
class SpaceShip: public GameObject {
private:
  typedef void (SpaceShip::*HitFunctionPtr)(GameObject&);
  typedef map<string, HitFunctionPtr> HitMap;
  ...
};
SpaceShip::HitFunctionPtr
SpaceShip::lookup(const GameObject& whatWeHit)
{
  static HitMap collisionMap;
  ...
}
```

此处，**collisionMap** 就是我们的映射表。它映射类名（一个 **string** 对象）到一个 **Spaceship** 的成员函数指针。因为 **map<string, HitFunctionPtr>**太拗口了，我们用了一个

类型重定义。（开个玩笑，试一下不用 **HitMap** 和 **HitFunctionPtr** 这两个类型重定义来写 **collisionMap** 的申明。大部分人不会做第二次的。）

给出了 **collisionMap** 后，**lookup** 的实现有些虎头蛇尾。因为搜索工作是 **map** 类直接支持的操作，并且我们在 **typeid()**的返回结果上总可以调用的（可移植的）一个成员函数是 **name()**（可以确定（注 **11**），它返回对象的动态类型的名字）。于是，实现 **lookup**，仅仅是根据形参的动态类型在 **collisionMap** 中找到它的对应项、

lookup 的代码很简单，但如果不熟悉标准模板库的话（再次参见 **Item M35**），就不会怎么简单了。别担心，程序中的注释解释了每一步在做什么。

```
SpaceShip::HitFunctionPtr
SpaceShip::lookup(const GameObject& whatWeHit)
{
  static HitMap collisionMap;        // we'll see how to
                                     // initialize this below
  // look up the collision-processing function for the type
  // of whatWeHit. The value returned is a pointer-like
  // object called an "iterator" (see Item 35).
  HitMap::iterator mapEntry=
    collisionMap.find(typeid(whatWeHit).name());
  // mapEntry == collisionMap.end() if the lookup failed;
  // this is standard map behavior. Again, see Item 35.
  if (mapEntry == collisionMap.end()) return 0;
  // If we get here, the search succeeded. mapEntry
  // points to a complete map entry, which is a
  // (string, HitFunctionPtr) pair. We want only the
  // second part of the pair, so that's what we return.
  return (*mapEntry).second;
}
```

最后一句是 **return (*mapEntry).second** 而不是习惯上的 **mapEntry->second** 以满足 **STL** 的奇怪行为。具体原因见 **Item M18**。

I 初始化模拟虚函数表

现在来看 **collisionMap** 的初始化。我们很想这么做：

```
// An incorrect implementation
SpaceShip::HitFunctionPtr
SpaceShip::lookup(const GameObject& whatWeHit)
```

```
{
  static HitMap collisionMap;
  collisionMap["SpaceShip"] = &hitSpaceShip;
  collisionMap["SpaceStation"] = &hitSpaceStation;
  collisionMap["Asteroid"] = &hitAsteroid;
  ...
}
```

但，这将在每次调用 **lookup** 时都将成员函数指针加入了 **collisionMap**，这是不必要的开销。而且它不会编译通过，不过这是将要讨论的第二个问题。

我们需要的是只将成员函数指针加入 **collisionMap** 一次，在 **collisionMap** 构造时。这很容易完成；我们只需写一个私有的静态成员函数 **initializeCollisionMap** 来构造和初始化我们的映射表，然后用其返回值来初始化 **collisionMap**：

```
class SpaceShip: public GameObject {
private:
  static HitMap initializeCollisionMap();
  ...
};
SpaceShip::HitFunctionPtr
SpaceShip::lookup(const GameObject& whatWeHit)
{
  static HitMap collisionMap = initializeCollisionMap();
  ...
}
```

不过这意味着我们要付出拷贝赋值的代价（见 **Item M19** 和 **M20**）。我们不想这么做。如果 **initializeCollisionMap()**返回指针的话，我们就不需要付出这个代价，但这样就需要担心指针指向的 **map** 对象是否能在恰当的时候被析构了。

幸好，有个两全的方法。我们可以将 **collisionMap** 改为一个灵巧指针（见 **Item M28**）它将在自己被析构时 **delete** 所指向的对象。实际上，标准 **C++**运行库提供的模板 **auto_ptr**，正是这样的一个灵巧指针（见 **Item M9**）。通过将 **lookup** 中的 **collisionMap** 申明为 **static** 的 **auto_ptr**，我们可以让 **initializeCollisionMap** 返回一个指向初始化了的 **map** 对象的指针了，不用再担心资源泄漏了；**collisionMap** 指向的 **map** 对象将在 **collisinMap** 自己被析构时自动析构。于是：

```
class SpaceShip: public GameObject {
private:
```

```
  static HitMap * initializeCollisionMap();
  ...
};
SpaceShip::HitFunctionPtr
SpaceShip::lookup(const GameObject& whatWeHit)
{
  static auto_ptr<HitMap>
    collisionMap(initializeCollisionMap());
  ...
}
```

实现 **initializeCollisionMap** 的最清晰的方法看起来是这样的：

```
SpaceShip::HitMap * SpaceShip::initializeCollisionMap()
{
  HitMap *phm = new HitMap;
  (*phm)["SpaceShip"] = &hitSpaceShip;
  (*phm)["SpaceStation"] = &hitSpaceStation;
  (*phm)["Asteroid"] = &hitAsteroid;
  return phm;
}
```

但和我在前面指出的一样，这不能编译通过。因为 **HitMap** 被申明为包容一堆指向成员函数的指针，它们全带同样的参数类型，也就是 **GameObject**。但，**hitSpaceShip** 带的是一个 **spaceShip** 参数，**hitSpaceStation** 带的是 **SpaceStation**，**hitAsteroid** 带的是 **Asteroid**。虽然 **SpaceShip**、**SpaceStation** 和 **Asteroid** 能被隐式的转换为 **GameObject**，但对带这些参数类型的函数指针可没有这样的转换关系。

为了摆平你的编译器，你可能想使用 **reinterpret_casts**（见 **Item M2**），而它在函数指针的类型转换中通常是被舍弃的：

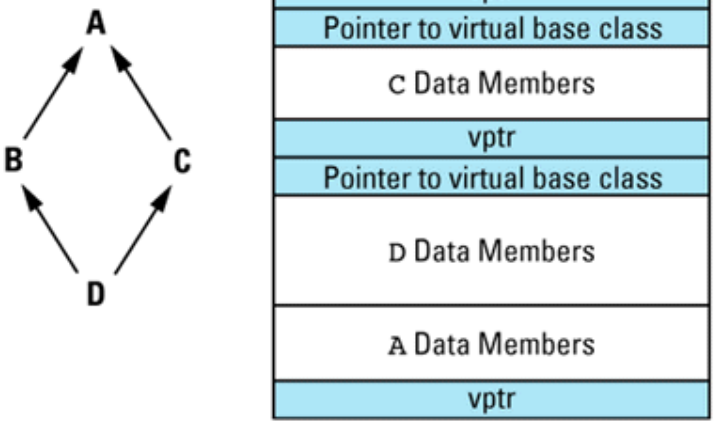
```
// A bad idea...
SpaceShip::HitMap * SpaceShip::initializeCollisionMap()
{
  HitMap *phm = new HitMap;
  (*phm)["SpaceShip"] =
    reinterpret_cast<HitFunctionPtr>(&hitSpaceShip);
  (*phm)["SpaceStation"] =
    reinterpret_cast<HitFunctionPtr>(&hitSpaceStation);
```

```
  (*phm)["Asteroid"] =
    reinterpret_cast<HitFunctionPtr>(&hitAsteroid);
  return phm;
}
```

这样可以编译通过，但是个坏主意。它必然伴随一些你绝不该做的事：对你的编译器撒谎。告诉编译器，**hitSpaceShip**、**hitSpaceStation** 和 **hitAsteroid** 期望一个 **GameObject** 类型的参数，而事实不是这样的。**hitSpaceShip** 期望一个 **SpaceShip**，**hitSpaceStation** 期望一个 **SpaceStation**，**hitAsteroid** 期望一个 **Asteroid**。这些 **cast** 说的是其它东西，它们撒谎了。

不只是违背了原则，这儿还有危险。编译器不喜欢被撒谎，当它们发现被欺骗后，它们经常会找出一个报复的方法。这此处，它们很可能通过产生错误的代码来报复你，当你通过***phm** 调用函数，而相应的 **GameObject** 的派生类是多重继承的或有虚基类时。如果 **SpaceStation**。**SpaceShip** 或 **Asteroid** 除了 **GameObject** 外还有其它基类，你可能会发现当你调用你在这儿搜索到的碰撞处理函数时，其行为非常的粗暴。

再看一下 **Item M24** 中描述的 **A—B—C—D** 的继承体系以及 **D** 的对象的内存布局。



D 中的四个类的部分，其地址都不同。这很重要，因为虽然指针和引用的行为并不相同（见 **Item M1**），编译器产生的代码中通常是通过指针来实现引用的。于是，传引用通常是通过传指针来实现的。当一个有多个基类的对象（如 **D** 的对象）传引用时，最重要的就是编译器要传递正确的地址——匹配于被调函数申明的形参类型的那个。

但如果你对你的编译器撒谎说你的函数期望一个 GameObject 而实际上要的是一个 SpaceShip 或一个 SpaceStation 时，发生什么？编译器将传给你错误的地址，导致运行期错误。而且将非常难以定位错误的原因。有很多很好的理由说明为什么不建议使用类型转换，这是其中之一。

OK，不使用类型转换。但函数指针类型不匹配的还没解决只有一个办法：将所有的函数都改为接受 GameObject 类型：

```
class GameObject {                      // this is unchanged
public:
  virtual void collide(GameObject& otherObject) = 0;
  ...
};
class SpaceShip: public GameObject {
public:
  virtual void collide(GameObject& otherObject);
  // these functions now all take a GameObject parameter
  virtual void hitSpaceShip(GameObject& spaceShip);
  virtual void hitSpaceStation(GameObject& spaceStation);
  virtual void hitAsteroid(GameObject& asteroid);
  ...
};
```

我们基于虚函数解决二重调度问题的方法中，重载了叫 collide 的函数。现在，我们理解为什么这儿没有照抄而使用了一组成员函数指针。所有的碰撞处理函数都有着相同的参数类型，所以必要给它们以不同的名字。

现在，我们可以以我们一直期望的方式来写 initializeCollisionMap 函数了：

```
SpaceShip::HitMap * SpaceShip::initializeCollisionMap()
{
  HitMap *phm = new HitMap;
  (*phm)["SpaceShip"] = &hitSpaceShip;
  (*phm)["SpaceStation"] = &hitSpaceStation;
  (*phm)["Asteroid"] = &hitAsteroid;
  return phm;
}
```

很遗憾，我们的碰撞函数现在得到的是一个更基本的 CameObject 参数而不是期望中的派生类类型。要想得到我们所期望的东西，必须在每个碰撞函数开始处采用 dynamic_cast

（见 Item M2）：

```
void SpaceShip::hitSpaceShip(GameObject& spaceShip)
{
  SpaceShip& otherShip=
    dynamic_cast<SpaceShip&>(spaceShip);
  process a SpaceShip-SpaceShip collision;
}
void SpaceShip::hitSpaceStation(GameObject& spaceStation)
{
  SpaceStation& station=
    dynamic_cast<SpaceStation&>(spaceStation);
  process a SpaceShip-SpaceStation collision;
}
void SpaceShip::hitAsteroid(GameObject& asteroid)
{
  Asteroid& theAsteroid =
    dynamic_cast<Asteroid&>(asteroid);
  process a SpaceShip-Asteroid collision;
}
```

如果转换失败，**dynamic_cast** 会抛出一个 **bad_cast** 异常。当然，它们从不会失败，因为碰撞函数被调用时不会带一个错误的参数类型的。只是，谨慎一些更好。

▌ 使用非成员的碰撞处理函数

我们现在知道了怎么构造一个类似 **vtbl** 的映射表以实现二重调度的第二部分，并且我们也知道了怎么将映射表的实现细节封装在 **lookup** 函数中。因为这张表包含的是指向成员函数的指针，所以在增加新的 **GameObject** 类型时仍然需要修改类的定义，这还是意味着所有人都必须重新编译，即使他们根本不关心这个新的类型。例如，如果增加了一个 **Satellite** 类型，我们不得不在 **SpaceShip** 类中增加一个处理 **SpaceShip** 和 **Satellite** 对象间碰撞的函数。所有 **SpaceShip** 的用户不得不重新编译，即使他们根本不在乎 **Satellite** 对象的存在。这个问题将导致我们否决只使用虚函数来实现二重调度，解决方法是只需做小小的修改。

如果映射表中包含的指针指向非成员函数，那么就没有重编译问题了。而且，转到非成员的碰撞处理函数将让我们发现一个一直被忽略的设计上的问题，就是，应该在哪个类里处理不同类型的对象间的碰撞？在前面的设计中，如果对象 **1** 和对象 **2** 碰撞，而正巧对象 **1** 是 **processCollision** 的左边的参数，碰撞将在对象 **1** 的类中处理；如果对象 **2** 正巧是左边的参数，碰撞就在对象 **2** 的类中处理。这个有特别的含义吗？是不是这样更好些：类型 **A**

和类型 B 的对象间的碰撞应该既不在 A 中也不在 B 中处理，而在两者之外的某个中立的地方处理？

如果将碰撞处理函数从类里移出来，我们在给用户提供类定义的头文件时，不用带上任何碰撞处理函数。我们可以将实现碰撞处理函数的文件组织成这样：

```
#include "SpaceShip.h"
#include "SpaceStation.h"
#include "Asteroid.h"
namespace {                    // unnamed namespace — see below
  // primary collision-processing functions
  void shipAsteroid(GameObject& spaceShip,
                    GameObject& asteroid);
  void shipStation(GameObject& spaceShip,
                   GameObject& spaceStation);
  void asteroidStation(GameObject& asteroid,
                       GameObject& spaceStation);
  ...
  // secondary collision-processing functions that just
  // implement symmetry: swap the parameters and call a
  // primary function
  void asteroidShip(GameObject& asteroid,
                    GameObject& spaceShip)
  { shipAsteroid(spaceShip, asteroid); }
  void stationShip(GameObject& spaceStation,
                   GameObject& spaceShip)
  { shipStation(spaceShip, spaceStation); }
  void stationAsteroid(GameObject& spaceStation,
                       GameObject& asteroid)
  { asteroidStation(asteroid, spaceStation); }
  ...
  // see below for a description of these types/functions
  typedef void (*HitFunctionPtr)(GameObject&, GameObject&);
  typedef map< pair<string,string>, HitFunctionPtr > HitMap;
  pair<string,string> makeStringPair(const char *s1,
                                     const char *s2);
```

```
  HitMap * initializeCollisionMap();
  HitFunctionPtr lookup(const string& class1,
                        const string& class2);
} // end namespace
void processCollision(GameObject& object1,
                      GameObject& object2)
{
  HitFunctionPtr phf = lookup(typeid(object1).name(),
                              typeid(object2).name());
  if (phf) phf(object1, object2);
  else throw UnknownCollision(object1, object2);
}
```

注意，用了无名的命名空间来包含实现碰撞处理函数所需要的函数。无名命名空间中的东西是当前编译单元（其实就是当前文件）私有的——很象被申明为文件范围内 **static** 的函数一样。有了命名空间后，文件范围内的 **static** 已经不赞成使用了，你应该尽快让自己习惯使用无名的命名空间（只要编译器支持）。

理论上，这个实现和使用成员函数的版本是相同的，只有几个轻微区别。第一，**HitFunctionPtr** 现在是一个指向非成员函数的指针类型的重定义。第二，意料之外的类 **CollisionWithUnknownObject** 被改叫 **UnknownCollision**，第三，其构造函数需要两个对象作参数而不再是一个了。这也意味着我们的映射需要三个消息了：两个类型名，一个 **HitFunctionPtr**。

标准的 **map** 类被定义为只处理两个信息。我们可以通过使用标准的 **pair** 模板来解决这个问题，**pair** 可以让我们将两个类型名捆绑为一个对象。借助 **makeStringPair** 的帮助，**initializeCollisionMap** 的实现如下：

```
// we use this function to create pair<string,string>
// objects from two char* literals. It's used in
// initializeCollisionMap below. Note how this function
// enables the return value optimization (see Item 20).
namespace {          // unnamed namespace again — see below
  pair<string,string> makeStringPair(const char *s1,
                                     const char *s2)
  { return pair<string,string>(s1, s2);   }
} // end namespace
namespace {          // still the unnamed namespace — see below
```

```
  HitMap * initializeCollisionMap()
  {
    HitMap *phm = new HitMap;
    (*phm)[makeStringPair("SpaceShip","Asteroid")] =
      &shipAsteroid;
    (*phm)[makeStringPair("SpaceShip", "SpaceStation")] =
      &shipStation;
  ...
  return phm;
}
} // end namespace
```

lookup 函数也必须被修改以处理 pair<string,string>对象，并将它作为映射表的第一部分：

```
namespace {          // I explain this below — trust me
  HitFunctionPtr lookup(const string& class1,
                        const string& class2)
  {
    static auto_ptr<HitMap>
      collisionMap(initializeCollisionMap());
    // see below for a description of make_pair
    HitMap::iterator mapEntry=
      collisionMap->find(make_pair(class1, class2));
    if (mapEntry == collisionMap->end()) return 0;
    return (*mapEntry).second;
  }
} // end namespace
```

这和我们以前写的代码几乎一样。唯一的实质性不同就是这个使用了 make_pair 函数的语句：

```
HitMap::iterator mapEntry=
  collisionMap->find(make_pair(class1, class2));
```

make_pair 只是标准运行库中的一个转换函数（模板）（见 Item E49 和 Item M35），它使得我们避免了在构造 pair 对象时需要申明类型的麻烦。我们本来要这样写的：

```
HitMap::iterator mapEntry=
  collisionMap->find(pair<string,string>(class1, class2));
```

这样写需要多敲好多字，而且为 **pair** 申明类型是多余的（它们就是 **class1** 和 **class2** 的类型），所以 **make_pair** 的形式更常见。

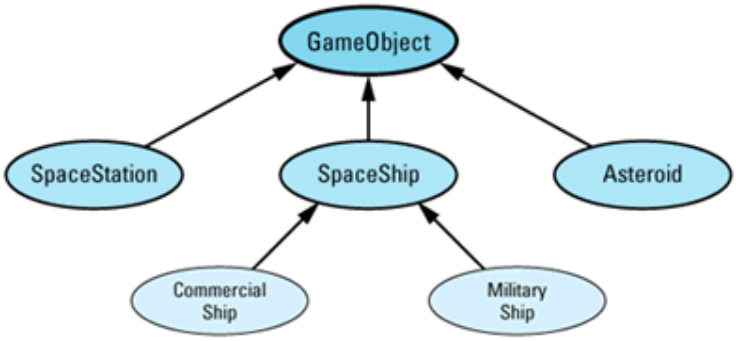
因为 **makeStringPair**、**initializeCollisionMap** 和 **lookup** 都是申明在无名的命名空间中的，它们的实现也必须在同一命名空间中。这就是为什么这些函数的实现在上面被写在了一个无名命名空间中的原因（必须和它们的申明在同一编译单元中）：这样链接器才能正确地将它们的定义（或说实现）与它们的前置申明关联起来。

我们最终达到了我们的目的。如果增加了新的 **GaemObject** 的子类，现存类不需要重新编译（除非它们用到了新类）。没有了 **RTTI** 的混乱和 **if...then...else** 的不可维护。增加一个新类只需要做明确定义了的局部修改：在 **initializeCollisionMap** 中增加一个或多个新的映射关系，在 **processCollision** 所在的无名的命名空间中申明一个新的碰撞处理函数。我们花了很大的力气才走到这一步，但至少努力是值得的。是吗？是吗？

也许吧。

I 继承与模拟虚函数表

我们还有最后一个问题需要处理。（如果，此时你奇怪老有最后一个问题要处理，你将认识到设计一个虚函数体系的难度。）我们所做的一切将工作得很好，只要我们不需要在调用碰撞处理函数时进行向基类映射的类型转换。假设我们开发的这个游戏某些时刻必须区分贸易飞船和军事飞船，我们将对继承体系作如下修改，根据 **Item M33** 的原则，将实体类 **CommercialShip** 和 **MilitaryShip** 从抽象类 **SpaceShip** 继承。



假设贸易飞船和军事飞船在碰撞过程中的行为是一致的。于是，我们期望可以使用相同的碰撞处理函数（在增加这两类以前就有的那个）。尤其是，在一个 **MilitaryShip** 对象和一个 **Asteroid** 对象碰撞时，我们期望调用

```
void shipAsteroid(GameObject& spaceShip,
                  GameObject& asteroid);
```

它不会被调用的。实际上，抛了一个 **UnknownCollision** 的异常。因为 **lookup** 在根据类型名“**MilitaryShip**”和“**Asteroid**”在 **collisionMap** 中查找函数时没有找到。虽然

MilitaryShip 可以被转换为一个 SpaceShip，但 lookup 却不知道这一点。

而且，没有没有一个简单的办法来告诉它。如果你需要实现二重调度，并且需要这儿的向上类型映射，你只能采用我们前面讨论的二次虚函数调用的方法（同时也意味着增加新类的时候，所有人都必须重新编译）。

I 初始化模拟虚函数表（再次讨论）

这就是关于二重调度的所有要说的，但是，用如此悲观的条款来结束是令人很不愉快的。因此，让我们用概述初始化 collisionMap 的两种方法来结束。

按目前情况来看，我们的设计完全是静态的。每次我们注册一个碰撞处理函数，我们就不得不永远留着它。如果我们想在游戏运行过程中增加、删除或修改碰撞处理函数，将怎么样？不提供。

但是是可以做到的。我们可以将映射表放入一个类，并由它提供动态修改映射关系的成员函数。例如：

```
class CollisionMap {
public:
  typedef void (*HitFunctionPtr)(GameObject&, GameObject&);
  void addEntry(const string& type1,
                const string& type2,
                HitFunctionPtr collisionFunction,
                bool symmetric = true);              // see below
  void removeEntry(const string& type1,
                   const string& type2);
  HitFunctionPtr lookup(const string& type1,
                        const string& type2);
  // this function returns a reference to the one and only
  // map — see Item 26
  static CollisionMap& theCollisionMap();
private:
  // these functions are private to prevent the creation
  // of multiple maps — see Item 26
  CollisionMap();
  CollisionMap(const CollisionMap&);
};
```

这个类允许我们在映射表中进行增加和删除操作，以及根据类型名对查找相应的碰撞处理函数。它也使用了 Item E26 中讲的技巧来限制 CollisionMap 对象的个数为 1，因为我

们的系统中只有一个映射表。（更复杂的游戏需要多张映射表是可以想象到的。）最后，它允许我们简化在映射表中增加对称性的碰撞（也就是说，类型 **T1** 的对象撞击 **T2** 的对象和 **T2** 的对象撞击 **T1** 的对象，其效果是相同的。）的过程，它自动增加对称的映射关系，只要 **addEntry** 被调用时可选参数 **symmetric** 被设为 **true**。

借助于 **CollisionMap** 类，每个想增加映射关系的用户可以直接这么做：

```
void shipAsteroid(GameObject& spaceShip,
                  GameObject& asteroid);
CollisionMap::theCollisionMap().addEntry("SpaceShip",
                                         "Asteroid",
                                         &shipAsteroid);
void shipStation(GameObject& spaceShip,
                 GameObject& spaceStation);
CollisionMap::theCollisionMap().addEntry("SpaceShip",
                                         "SpaceStation",
                                         &shipStation);
void asteroidStation(GameObject& asteroid,
                     GameObject& spaceStation);
CollisionMap::theCollisionMap().addEntry("Asteroid",
                                         "SpaceStation",
                                         &asteroidStation);
...
```

必须确保在发生碰撞前就将映射关系加入了映射表。一个方法是让 **GameObject** 的子类在构造函数中进行确认。这将导致在运行期的一个小小的性能开销。另外一个方法是创建一个 **RegisterCollisionFunction** 类：

```
class RegisterCollisionFunction {
public:
  RegisterCollisionFunction(
          const string& type1,
          const string& type2,
          CollisionMap::HitFunctionPtr collisionFunction,
          bool symmetric = true)
  {
    CollisionMap::theCollisionMap().addEntry(type1, type2,
                                             collisionFunction,
```

```
                                        symmetric);
  }
};
```

用户于是可以使用此类型的一个全局对象来自动地注册他们所需要的函数：

```
RegisterCollisionFunction cf1("SpaceShip", "Asteroid",
                              &shipAsteroid);
RegisterCollisionFunction cf2("SpaceShip", "SpaceStation",
                              &shipStation);
RegisterCollisionFunction cf3("Asteroid", "SpaceStation",
                              &asteroidStation);
...
int main(int argc, char * argv[])
{
  ...
}
```

因为这些全局对象在 **main** 被调用前就构造了，它们在构造函数中注册的函数也在 **main** 被调用前就加入映射表了。如果以后增加了一个派生类

```
class Satellite: public GameObject { ... };
```

以及一个或多个碰撞处理函数

```
void satelliteShip(GameObject& satellite,
                   GameObject& spaceShip);
void satelliteAsteroid(GameObject& satellite,
                       GameObject& asteroid);
```

这些新函数可以用同样方法加入映射表而不需要修改现存代码：

```
RegisterCollisionFunction cf4("Satellite", "SpaceShip",
                              &satelliteShip);
RegisterCollisionFunction cf5("Satellite", "Asteroid",
                              &satelliteAsteroid);
```

这不会改变实现多重调度没有完美解决方法的事实。但它使得容易提供数据给基于 **map** 的实现，如果我们认为这种实现最接近我们的需要的话。

I 注 11：

要指出的是，不是那么可完全确定的。**C++**标准并没有规定 **type_info::name** 的返回值，不同的实现，其行为会有区别。（例如，对于类 **Spaceship**，**type_info::name** 的一个实现返回“**class SpaceShip**”。）更好的设计是通过它所关联的 **type_info** 对象的地址了鉴别一个

类，因为每个类关联的 type_info 对象肯定是不同的。HitMap 于是应该被申明为 map<cont type_info *, HitFunctionPtr>。

8. 杂项

我们现在到了接近结束的部分了，这章讲述的是一些不属于前面任一章节的指导原则。开始两个是关于 C++软件开发的，描述的是设计适应变化的系统。面向对象的一个强大之处是支持变化，这两个条款描述具体的步骤来增强你的软件对变化的抵抗能力。

然后，我们分析怎么在同一程序中进行 C 和 C++混合编程。这必然导致考虑语言学之外的问题，但 C++存在于真实世界中，所以有时我们必须面对这种事情。

最后，我将概述 C++语言标准在《The Annotated C++ Reference Manual 》出版后发生的变化。尤其是，我将覆盖标准运行库中发生的广大变化（参见 Item E49）。如果你没有紧跟标准化的过程，那么将可能很吃惊——很多变化都令人很开心。

8.1 Item M32：在未来时态下开发程序

事物在变化。

作为软件开发人员，我们也许知道得不多，但我们知道万物都会变化。我们没必要知道什么将发生变化，这么变化又怎么发生，以什么时候发生，在哪里发生，但我们知道：万物都会变化。

好的软件能够适应变化。它提供新的特性，适应到新的平台，满足新的需求，处理新的输入。软件的灵活性、健壮性、可靠性不是来自于意外。它是程序员们在满足了现在的需求并关注了将来的可能后设计和实现出来的。这样的软件（接受小改变的软件）是那些在未来时态下开发程序的人写出来的。

要在未来时态下开发程序，就必须接受事物会发生变化，并为此作了准备。这是应该考虑的：新的函数将被加入到函数库中，新的重载将发生，于是要注意那些含糊的函数调用行为的结果；新的类将会加入继承层次，现在的派生类将会是以后的基类，并已为此作好准备；将会编制新的应用软件，函数将在新的运行环境下被调用，它们应该被写得在新平台上运行正确；程序的维护人员通常不是原来编写它们的人，因此应该被设计得易于被别人理解、维护和扩充。

这么做的一种方法是：用 C++语言自己来表达设计上的约束条件，而不是用注释或文档。例如，如果一个类被设计得不会被继承，不要只是在其头文件中加个注释，用 C++的方法来阻止继承；Item M26 显示了这个技巧。如果一个类需要其实例全部创建在堆中，不要只是对用户说了这么一句，用 Item M27 的方法来强迫这一点。如果拷贝构造和赋值对一个类是没有意义的，通过申明它们为私有来阻止这些操作（见 Item E27）。C++提供了强大的功能、

灵活度和表达力。用语言提供的这些特性来强迫程序符合设计。

因为万物都会变化，要写能承受软件发展过程中的混乱攻击的类。避免“**demand-paged**”（**WQ**：“用户要求型”之类的意思吧）的虚函数，凭什么你本没有写虚函数而直到有人来要求后你就更改为虚函数？应该判断一个函数的含意，以及它被派生类重定义的话是否有意义。如果是有意义的，申明它为虚，即使没有人立即重定义它。如果不是的话，申明它为非虚，并且不要在以后为了便于某人而更改；确保更改是对整个类的运行环境和类所表示的抽象是有意义的（见 **Item E36**）。

处理每个类的赋值和拷贝构造函数，即使“从没人这样做过”。他们现在没有这么做并不意味着他们以后不这么做（见 **Item E18**）。如果这些函数是难以实现的，那么申明它们为私有。这样，不会有人误调编译器提供的默认版本而做错事（这在默认赋值和拷贝构造函数上经常发生，见 **Item E11**）。

基于最小惊讶法则：努力提供这样的类，它们的操作和函数有自然的语法和直观的语义。和内建数据类型的行为保持一致：拿不定主意时，仿照 **int** 来做。

要承认：只要是能被人做的，就有人这么做（**WQ**：莫菲法则）。他们会抛异常；会用自己给自己赋值；在没有赋初值前就使用对象；给对象赋了值而没有使用；会赋过大的值、过小的值或空值。一般而言，只要能编译通过，就有人会这么做。所以，要使得自己的类易于被正确使用而难以误用。要承认用户可能犯错误，所以要将你的类设计得可以防止、检测或修正这些错误（例子见 **Item M33** 和 **Item E46**）。

努力于可移植的代码。写可移植的代码并不比不可移植的代码难太多，只有在性能极其重要时采用不可移植的结构才是可取的（见 **Item M16**）。即使是为特定的硬件设计的程序也经常被移植，因为这些平台在几年内就会有一个数量级的性能提升。可移植的代码使得你在更换平台是比较容易，扩大你的用户基础，吹嘘支持开放平台。这也使得你赌错了操作系统时比较容易补救。

将你的代码设计得当需要变化时，影响是局部的。尽可能地封装；将实现细节申明为私有（例子见 **Item E20**）。只要可能，使用无名的命名空间和文件内的静态对象或函数（见 **Item E31**）。避免导致虚基类的设计，因为这种类需要每个派生类都直接初始化它——即使是那些间接派生类（见 **Item M4** 和 **Item E43**）。避免需要 **RTTI** 的设计，它需要 **if...then...else** 型的瀑布结构（再次参见 **Item M31**，然后看 **Item E39** 上的好方法）。每次，类的继承层次变了，每组 **if...then...else** 语句都需要更新，如果你忘掉了一个，你不会从编译器得到任何告警。

这是著名的老生常谈般的告戒，但大部分程序员仍然违背它。看这条一个著名 **C++**专家提出忠告（很不幸，许多作者也这么说）：

你需要虚析构函数，只要有人 ***delete*** *一个实际值向* ***D*** *的* ***B **** *。*

这里，**B** 是基类，**D** 是其派生类。换句话说，这位作者暗示，如果你的程序看起来是这

样时，并不需要 B 有虚析构函数：

```
class B { ... };                    // no virtual dtor needed
class D: public B { ... };
B *pb = new D;
```

然而，当你加入这么一句时，情况就变了：

```
delete pb;                          // NOW you need the virtual
                                    // destructor in B
```

这意味着，用户代码中的一个小变化——增加了一个 **delete** 语句——实际上能导致需要修改 B 的定义。如果这发生了的话，所有 B 的用户都必须重编译。采纳了这个作者的建议的话，一条语句的增加将导致大量代码的重编译和重链接。这绝不是一个高效的设计。

就同一主题，另一个作者写道：

如果一个公有基类没有虚析构函数，所有的派生类基其成员函数都不应该有析构函数。

也就是说，这是没问题的：

```
class string {                      // from the standard C++ library
public:
  ~string();
};
class B { ... };                    // no data members with dtors,
                                    // no virtual dtor needed
```

但从 B 继承一个新类后，事情就变了：

```
class D: public B {
  string name;                      // NOW ~B needs to be virtual
};
```

再一次，一个关于 B 的使用的小小的变化（这里是增加了一个包含有析构函数的成员对象的派生类）可能需要大量代码的重编译和重链接。但在系统中，小的变化应该只有小的影响。这个设计在这个测试上失败了。

同一作者写了：

如果多重继承体系有许多析构函数，每个基类都应该有应该虚析构函数。

所有这些引用，都在关注进行时态的考虑。用户现在在怎么操纵指针？当前类的什么成员有析构函数？继承系统中的什么类有析构函数？

未来时态的考虑完全不同。不是问一个类现在正被怎么使用，而是问这个类是被设计为怎么去使用的。未来时态的考虑认为：如果一个类被设计为作一个基类使用（即使现在还没有被这么使用），它就应该有一个虚析构函数（见 **Item E14**）。这样的类在现在和将来都行为正确，并且当新类从它们派生时并不影响其它库用户。（至少，它们没有任何影响，直

到其析构函数被使用。如果需要对类的额外修改，其它用户将受影响。）

有一个商业类库（在 C++标准运行库申明 string 以前）包含了一个没有虚析构函数的 string 类。其生产商解释：

*我们没有使用虚析构函数，因为我们不想这个 **string** 类有 **vtbl**。我们甚至不期望想有一个 **string ***，所以这不成为问题。我们很清楚这么做将造成的困难。*

这是进行时态的考虑还是未来时态的考虑？

当然，vbtl 有个技术上的问题（见 Item M24 和 Item E14）。大部分 string 类的实现都是对象内部只有一个 char *指针，所以增加一个 vptr 造成每个 string 类的大小翻倍。所以可以理解为什么不肯实现它，尤其是对于 string 这样频繁出现高密度使用的类。这样的类是属于影响程序性能的那 20%的部分的（见 Item M16）。

还有，一个 string 对象的全部内存——它自己加上所指向的在堆中的字符串——通常远大于保存一个 char *指针的大小。从这个方面来说，为 vptr 增加的花费并不是那么重要的。不过，这仍然是个合法的设计。（的确，ISO/ANSI 标准委员会似乎是这么想的：标准 string 类型有一个非虚的析构函数。）

更有问题的是生产商的注释：“我们甚至不期望有一个 string *，所以这不成为问题”。这可能是正确的，但他们的这个 string 类是提供给数以千记的开发人员使用的类库的一部分。有如此多的开发人员，每个人对 C++的掌握程度都不同，每个人做事的方法也都不同。这些人都了解 string 没有虚析构函数的后果吗？生产商确信他们的客户知道没有虚析构函数时，用 string *指针 delete 对象时可能工作不正确，在 string 的指针和引用上使用 RTTI 操作可能得到不正确的信息吗？这个类是易于正确使用而不容易用错的吗？

这个生产商应该提供明确的文档以指出他的 string 类没有被设计得可被继承的，但如果程序员没注意到这个警告或未能读到这个文档时会发生什么？

一个可选方法是用 C++自己来阻止继承。Item M26 描述了怎么限制对象只生成于堆中，以及用 auto_ptr 对象来操作堆中的对象。构造 string 对象的接口将不符合传统也不方便，需要这样：

```
auto_ptr<String> ps(String::makeString("Future tense C++"));
...                                  // treat ps as a pointer to
                                     // a String object, but don't
                                     // worry about deleting it
```

来代替这个：

```
String s("Future tense C++");
```

但，多半，为了减少不正确的继承行为是值得换用不方便的语法的。（对 string 类，这未必很合算，但对其它类，这样的交换是完全值得的。）

当然，进行时态的考虑也是需要的。你开发的软件必须在现在的编译器下工作；你不

能等到直到最新的语言特性（被编译器）实现了。它必须在现在支持的硬件上工作，也必须在你的用户已有的（软件）配置下工作；你不能强迫客户升级系统或更改操作环境。它必须现在就提供可接受的性能；承诺数年后更小而更快的程序完全不能吸引潜在用户。你所参与的软件必须“尽快”推出，通常意味着已经误期了（**which often means some time in the recent past**）。这些都是重要的约束条件。你不能忽略它们。

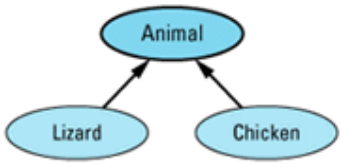
未来时态的考虑只是简单地增加了一些额外约束：

- 提供完备的类（见 **Item E18**），即使某些部分现在还没有被使用。如果有了新的需求，你不用回过头去改它们。
- 将你的接口设计得便于常见操作并防止常见错误（见 **Item E46**）。使得类容易正确使用而不易用错。例如，阻止拷贝构造和赋值操作，如果它们对这个类没有意义的话（见 **Item E27**）。防止部分赋值（见 **Item M33**）。
- 如果没有限制你不能通用化你的代码，那么通用化它。例如，如果在写树的遍历算法，考虑将它通用得可以处理任何有向不循环图。

未来时态的考虑增加了你的代码的可重用性、可维护性、健壮性，已及在环境发生改变时易于修改。它必须与进行时态的约束条件进行取舍。太多的程序员们只关注于现在的需要，然而这么做牺牲了其软件的长期生存能力。是与众不同的，是离经叛道的，在未来时态下开发程序。

8.2 Item M33：将非尾端类设计为抽象类

假设你正在从事一个软件项目，它处理动物。在这个软件里，大多数动物能被抽象得非常类似，但两种动物——晰蜴和小鸡——需要特别处理。显然，晰蜴和小鸡与动物类的联系是这样的：



动物类处理所有动物共有的特性，晰蜴类和小鸡类特别化动物类以适用这两种动物的特有行为。

这是它们的简化定义：

```
class Animal {
public:
  Animal& operator=(const Animal& rhs);
  ...
```

```
};
class Lizard: public Animal {
public:
  Lizard& operator=(const Lizard& rhs);
  ...
};
class Chicken: public Animal {
public:
  Chicken& operator=(const Chicken& rhs);
  ...
};
```

这里只写出了赋值运算函数，但已经够我们忙乎一阵了。看这样的代码：

```
Lizard liz1;
Lizard liz2;
Animal *pAnimal1 = &liz1;
Animal *pAnimal2 = &liz2;
...
*pAnimal1 = *pAnimal2;
```

这里有两个问题。第一，最后一行的赋值运算调用的是 **Animal** 类的，虽然相关对象的类型是 **Lizard**。结果，只有 **liz1** 的 **Animal** 部分被修改。这是部分赋值。在赋值后，**liz1** 的 **Animal** 成员有了来自于 **liz2** 的值，但其 **Lizard** 成员部分没被改变。

第二个问题是真的有程序员把代码写成这样。用指针来给对象赋值并不少见，特别是那些对 **C** 有丰富经验而转移到 **C++**的程序员。所以，我们应该将赋值设计得更合理的。如 **Item M32** 指出的，我们的类应该容易被正确适用而不容易被用错，而上面这个类层次是容易被用错。

一个解决方法是将赋值运算申明为虚函数。如果 **Animal::operator=**是虚函数，那句赋值语句将调用 **Lizard** 的赋值操作（应该被调用的版本）。然而，看一下申明它为虚后会发生什么：

```
class Animal {
public:
  virtual Animal& operator=(const Animal& rhs);
  ...
};
class Lizard: public Animal {
```

```
public:
  virtual Lizard& operator=(const Animal& rhs);
  ...
};
class Chicken: public Animal {
public:
  virtual Chicken& operator=(const Animal& rhs);
  ...
};
```

基于C++语言最近作出的修改，我们可以修改返回值的类型（于是每个都返回正确的类的引用），但C++的规则强迫我们申明相同的参数类型。这意味着Lizard类和Chicken类的赋值操作必须准备接受任意类型的Animal对象。也就是说，这意味着我们必须面对这样的事实：下面的代码是合法的：

```
Lizard liz;
Chicken chick;
Animal *pAnimal1 = &liz;
Animal *pAnimal2 = &chick;
...
*pAnimal1 = *pAnimal2;                  // assign a chicken to
                                        // a lizard!
```

这是一个混合类型赋值：左边是一个Lizard，右边是一个Chicken。混合类型赋值在C++中通常不是问题，因为C++的强类型原则将评定它们非法。然而，通过将Animal的赋值操作设为虚函数，我们打开了混合类型操作的门。

这使得我们处境艰难。我们应该允许通过指针进行同类型赋值，而禁止通过同样的指针进行混合类型赋值。换句话说，我们想允许这样：

```
Animal *pAnimal1 = &liz1;
Animal *pAnimal2 = &liz2;
...
*pAnimal1 = *pAnimal2;                  // assign a lizard to a lizard
```

而想禁止这样：

```
Animal *pAnimal1 = &liz;
Animal *pAnimal2 = &chick;
...
*pAnimal1 = *pAnimal2;                  // assign a chicken to a lizard
```

只能在运行期区分它们，因为将*pAnimal2 赋给*pAnimal1 有时是正确的，有时不是。我们于是陷入了基类型运行期错误的黑暗世界中。尤其是，我们需要在混合类型赋值时指出在 operator=内部发生了错误，而类型相同时，我们期望按通常的方式完成赋值。

我们可以使用 dynamic_cast（见 Item M2）来实现。下面是怎么实现 Lizard 的赋值操作：

```
Lizard& Lizard::operator=(const Animal& rhs)
{
  // make sure rhs is really a lizard
  const Lizard& rhs_liz = dynamic_cast<const Lizard&>(rhs);
  proceed with a normal assignment of rhs_liz to *this;
}
```

这个函数只在 rhs 确实是 Lizard 类型时将它赋给*this。如果 rhs 不是 Lizard 类型，函数传递出 dynamic_cast 转换失败时抛的 bad_cast 类型的异常。（实际上，异常的类型是 std::bad_cast，因为标准运行库的组成部分，包括它们抛出的异常，都位于命名空间 std 中。对于标准运行库的概述，见 Item E49 和 Item M35）。

即使不在乎有异常，这个函数看起来也是没必要的复杂和昂贵——dynamic_cast 必要引用一个 type_info 结构；见 Item M24——因为通常情况下都是一个 Lizard 对象赋给另一个：

```
Lizard liz1, liz2;
...
liz1 = liz2;                          // no need to perform a
                                      // dynamic_cast: this
                                      // assignment must be valid
```

我们可以处理这种情况而无需增加复杂度或花费 dynamic_cast，只要在 Lizard 中增加一个通常形式的赋值操作：

```
class Lizard: public Animal {
public:
  virtual Lizard& operator=(const Animal& rhs);
  Lizard& operator=(const Lizard& rhs);          // add this
  ...
};
Lizard liz1, liz2;
...
liz1 = liz2;                                  // calls operator= taking
```

```
                                                    // a const Lizard&
Animal *pAnimal1 = &liz1;
Animal *pAnimal2 = &liz2;
...
*pAnimal1 = *pAnimal2;                          // calls operator= taking
                                                    // a const Animal&
```

实际上，给出了后面那个的 **operator=**，也就简化了前者的实现：

```
Lizard& Lizard::operator=(const Animal& rhs)
{
  return operator
}
```

现在这个函数试图将 **rhs** 转换为一个 **Lizard**。如果转换成功，通常的赋值操作被调用；否则，一个 **bad_cast** 异常被抛出。

说实话，在运行期使用 **dynamic_cast** 进行类型检测，这令我很紧张。有一件事要注意，一些编译器仍然没有支持 **dynamic_cast**，所以使用它的代码虽然理论上具有可移植性，实际上不一定。更重要的是，它要求使用 **Lizard** 和 **Chicken** 的用户必须在每次赋值操作时都准备好捕获 **bad_cast** 异常并作相应处理。如果他们没有这么做的话，那么不清楚我们得到的好处是否超过最初的方案。

指出了这个关于虚赋值操作的令人非常不满意的状态后，在最开始的地方重新整理以试图找到一个方法来阻止用户写出有问题的赋值语句是有必要的。如果这样的赋值语句在编译期被拒绝，我们就不用担心它们做错事了。

最容易的方法是在 **Animal** 中将 **operator=**置为 **private**。于是，**Lizard** 对象可以赋值给 **Lizard** 对象，**Chicken** 对象可以赋值给 **Chicken** 对象，但部分或混合类型赋值被禁止：

```
class Animal {
private:
  Animal& operator=(const Animal& rhs);              // this is now
  ...                                                // private
};
class Lizard: public Animal {
public:
  Lizard& operator=(const Lizard& rhs);
  ...
};
class Chicken: public Animal {
```

```
public:
  Chicken& operator=(const Chicken& rhs);
  ...
};
Lizard liz1, liz2;
...
liz1 = liz2;                                    // fine
Chicken chick1, chick2;
...
chick1 = chick2;                                // also fine
Animal *pAnimal1 = &liz1;
Animal *pAnimal2 = &chick1;
...
*pAnimal1 = *pAnimal2;                          // error! attempt to call
                                                //                private
Animal::operator=
```

不幸的是，**Animal** 也是实体类，这个方法同时将 **Animal** 对象间的赋值评定为非法了：

```
Animal animal1, animal2;
...
animal1 = animal2;                              // error! attempt to call
                                                //                private
Animal::operator=
```

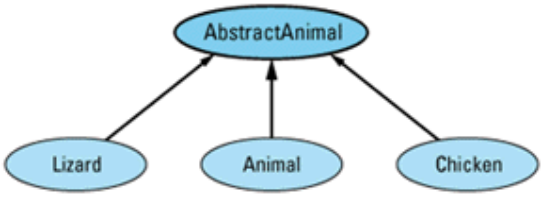
而且，它也使得不可能正确实现 **Lizard** 和 **Chicken** 类的赋值操作，因为派生类的赋值操作函数有责任调用其基类的赋值操作函数：

```
Lizard& Lizard::operator=(const Lizard& rhs)
{
  if (this == &rhs) return *this;
  Animal::operator=(rhs);                       // error! attempt to call
                                                // private function. But
                                                // Lizard::operator= must
                                                // call this function to
  ...                                           // assign the Animal parts
}                                               // of *this!
```

后面这个问题可以通过将 **Animal::operator=**申明为 **protected** 来解决，但“允许

Animal 对象间的赋值而阻止 **Lizard** 和 **Chicken** 对象通过 **Animal** 的指针进行部分赋值”的两难问题仍然存在。程序该怎么办？

最容易的事情是排除 **Animal** 对象间赋值的需求，其最容易的实现方法是将 **Animal** 设计为抽象类。作为抽象类，**Animal** 不能被实例化，所以也就没有了 **Animal** 对象间赋值的需求了。当然，这导致了一个新问题，因为我们最初的设计表明 **Animal** 对象是必须的。有一个很容易的解决方法：不用将 **Animal** 设为抽象类，我们创一个新类——叫 **AbstractAnimal**——来包含 **Animal**、**Lizard**、**Chikcen** 的共有属性，并把它设为抽象类。然后将每个实体类从 **AbstractAnimal** 继承。修改后的继承体系是这样的：



类的定义是：

```
class AbstractAnimal {
protected:
  AbstractAnimal& operator=(const AbstractAnimal& rhs);
public:
  virtual ~AbstractAnimal() = 0;                    // see below
  ...
};
class Animal: public AbstractAnimal {
public:
  Animal& operator=(const Animal& rhs);
  ...
};
class Lizard: public AbstractAnimal {
public:
  Lizard& operator=(const Lizard& rhs);
  ...
};
class Chicken: public AbstractAnimal {
public:
  Chicken& operator=(const Chicken& rhs);
```

```
  ...
};
```

这个设计给你所以你需要的东西。同类型间的赋值被允许，部分赋值或不同类型间的赋值被禁止；派生类的赋值操作函数可以调用基类的赋值操作函数。此外，所有涉及 **Aniaml**、**Lizard** 或 **Chicken** 类的代码都不需要修改，因为这些类仍然操作，其行为与引入 **AbstractAnimal** 前保持了一致。肯定，这些代码需要重新编译，但这是为获得“确保了编译通过的赋值语句的行为是正确的而行为可能不正确的赋值语句不能编译通过”所付出的很小的代价。

要使得这一切工作，**AbstractAnimal** 类必须是抽象类——它必须至少有一个纯虚函数。大部分情况下，带一个这样的函数是没问题的，但在极少见的情况下，你会发现需要创一个如 **AbstractAnimal** 这样的类，没有哪个成员函数是自然的纯虚函数。此时，传统方法是将析构函数申明为纯虚函数；这也是上面所采用的。为了支持多态，基类总需要虚析构函数（见 **Item 14**），将它再多设为纯虚的唯一麻烦就是必须在类的定义之外实现它（例子见 **P195**，**Item M29**）。

（如果实现一个纯虚函数的想法冲击了你，你只是知识不够开阔。申明一个函数为虚并不意味着它没有实现，它意味着：

- 当前类是抽象类
- 任何从此类派生的实体类必须将此函数申明为一个“普通”的虚函数（也就是说，不能带“**= 0**”）

是的，绝大部分纯虚函数都没有实现，但纯虚析构函数是个特例。它们必须被实现，因为它们在派生类析构函数被调用时也将被调用。而且，它们经常执行有用的任务，诸如释放资源（见 **Item M9**）或纪录消息。实现纯虚函数一般不常见，但对纯虚析构函数，它不只是常见，它是必须。）

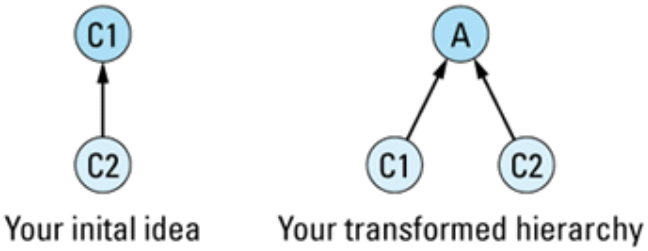
你可能已经注意到这里讨论的通过基类指针进行赋值的问题是基于假设实体类（如 **Animal**）有数据成员。如果它们没有数据成员，你可能指出，那么就不会有问题，从一个无数据的实体类派生新的实体类是安全的。

无数据而可以成为实体类的基类会两种可能：在将来，或者它可能有数据成员，或者它仍然没有。如果它将来可能有数据成员，你现在做的只是推迟问题的发生（直到数据成员被加入），你在用短利换长痛（参见 **Item M32**）。如果这个基类真的不会有数据成员，那么它现在就该是抽象类，没有数据的实体类有什么用处？

用如 **AbstractAnimal** 这样的抽象基类替换如 **Animal** 这样的实体基类，其好处远比简单地使得 **operator=**的行为易于了解。它也减少了你试图对数组使用多态的可能，这种行为的令人不愉快的后果解释于 **Item M3**。然而，这个技巧最大的好处发生在设计的层次上，因为这种替换强迫你明确地认可有用处的抽象行为的实体。也就是说，它使得你为有用的原型

（**concept**）创造了新的抽象类，即使你并不知道这个有用的原型的存在。

如果你有两个实体类 **C1** 和 **C2** 并且你喜欢 **C2** 公有继承自 **C1**，你应该将两个类的继承层次改为三个类的继承层次，通过创造一个新的抽象类 **A** 并将 **C1** 和 **C2** 都从它继承：



这种修改的重要价值是强迫你确定抽象类 **A**。很清楚，**C1** 和 **C2** 有共性；这就是为什么它们用公有继承联系在一起的原因（见 **Item E35**）。修改后，你必须确定这些共性到底是什么。而且，你必须用 **C++**的类将这些共性组织起来，它将不再是模糊不清的东西了，它到达了一个抽象类型的层次，有明确定义的成员函数和明确定义的语义。

这一切导致了一些令人不安的思考。毕竟，每个类都完成了某些类型的抽象，我们不应该在此继承体系中创造两个类来针对每个原型吗（一个是抽象类来表示其抽象部分**(to embody the abstract part of the abstraction)**，一个是实体类来表示对象生成部分**(to embody the object-generation part of the abstraction)**）？不应该。如果你这么做了，将使得继承体系中有太多的类。这样的继承体系是难以理解的，难以维护的，编译的代价很昂贵。这不是面向对象设计的目的。

其目的是：确认有用的抽象，并强迫它们（并且只有它们）放入如抽象类这样的实体。但怎么确认有用的抽象？谁知道什么抽象在将来被证明有用？谁能预知他将来要从什么进行继承？

好了，我不知道怎么预知一个继承体系将来的用处，但我知道一件事：在一个地方需要的抽象可能只是凑巧，但多处地方需要的抽象通常是有意义的。那么，有用的抽象就是那些被多处需要的抽象。也就是说，它们相当于是这样的类：就它们自己而言是有用的（比如，有这种类型的对象是用处的），并且它们对于一个或多个派生类也是有用处的。

在一个原型第一次被需要时，我们无法证明同时创造一个抽象类（为了这个原型）和一个实体类（为了原型对应的对象）是正确的，但第二次需要时，我们就能够这么做是正确的。我描述过的修改简单地实现了这个过程，并且在这么做的过程中强迫设计着和程序员明确表达那些有用的抽象，即使他们不知道那些有用的原型。 这也碰巧使得构建正确的赋值行为很容易。

让我们看一下一个简单的例子。假设你正在编制一个程序以处理局域网上计算机间的移动信息，通过将它拆为数据包并根据某种协议进行传输。我们认为应该用类来表示这些数

据数据包，并且这些数据包是程序的核心。

假设你处理的只有一种传输协议，也只有一种包。也许你听说了其它协议和数据包类型的存在，但还从未支持它们，也没有任何计划以在未来支持它们。你会为数据包（**for the concept that a packet represents**） 既设计一个抽象类吗，又设计一个你实际使用的实体类？如果你这么做了，你可以在以后增加新的数据包而不用改变基类。这使得你增加新的数据包类型时程序不用重新编译。但这种设计需要两个类，而你现在只需要一个（针对于你现在使用的特殊数据包类型）。这值得吗，增加设计的复杂度以允许扩充特性，而这种扩充可能从不发生？

这儿没有肯定正确的选择，但经验显示：为我们还不完全了解的原型设计优秀的类几乎是不可能的。如果你为数据包设计了抽象类，你怎么保证它正确，尤其是在你的经验只局限于这唯一的数据包类型时？记住，只有在设计出的类能被将来的类从它继承而不需要它作任何修改时，你才能从数据包的抽象类中获得好处。（如果它需要被修改，你不得不重新编译所有使用数据包类的代码，你没得到任何好处。）

看起来不太能够设计出一个领人满意的抽象设计包类，除非你精通各种数据包的区别以及它们相应的使用环境。鉴于你有限的经验，我建议不要定义抽象类，等到以后需要从实体类继承时再加。

我所说的转换方法是一个判断是否需要抽象类的方法，但不是唯有的方法。还有很多其它的好方法；讲述面向对象分析的书籍上满是这类方法。“当发现需求从一个实体类派生出另外一个实体类时”，这也不是唯一需要引入抽象类的地方。不管怎么说啦，需要通过公有继承将两个实体类联系起来，通常表示需要一个新的抽象类。

这种情况是如此常见，所以引起了我们的深思。第三方的 **C++**类库越来越多，当发现你需要从类库中的实体类派生出一个新的实体类，而这个库你只有只读权时，你要怎么做？

你不能修改类库以加入一个新的抽象类，所以你的选择将很有限、很无趣：

- 从已存在的实体类派生出你的实体类，并容忍我们在本 **Item** 开始时说到的赋值问题。你还要注意在 **Item M3** 中说过的数组问题。
- 试图在类库的继承树的更高处找到一个完成了你所需的大部分功能的抽象类，从它进行继承。当然，可能没有合适的类；即使有，你可能不得不重复很多已经在（你试图扩展的）实体类中实现了的东西。
- 用包容你试图继承的类的方法来实现你的新类（见 **Item E40** 和 **Item E42**）。例下例，你将一个类库中的类的对象为数据成员，并在你的类中重实现它的接口：

```
class Window {                          // this is the library class
public:
virtual void resize(int newWidth, int newHeight);
virtual void repaint() const;
```

```
int width() const;
int height() const;
};
class SpecialWindow {              // this is the class you
public:                            // wanted to have inherit
  ...                              // from Window
  // pass-through implementations of nonvirtual functions
  int width() const { return w.width(); }
  int height() const { return w.height(); }
  // new implementations of "inherited" virtual functions
  virtual void resize(int newWidth, int newHeight);
  virtual void repaint() const;
private:
    Window w;
};
```

这种方法需要你在类库每次升级时也要更新你自己的类。它还需要你放弃重定义类库中的类的虚函数的能力，因为你用的不是继承。

- 使用你得到。使用类库中的类，而将你自己的程序修改得那个类适用。用非成员函数来提供扩展功能（那些你想加入那个类而没有做到的）。结果，程序将不如你所期望中的清晰、高效、可维护、可扩展，但至少它完成了你所需要的功能。

这些选择都不怎么吸引人，所以你不得不作出判断并选择最轻的毒药。这不怎么有趣，但生活有时就是这样。想让事情在以后对你自己（和我们其它人）容易些，将问题反馈给类库生产商。靠运气（以及大量的用户反馈），随时间的流逝，那些设计可能被改进。

最后，一般的规则是：非尾端类应该是抽象类。在处理外来的类库时，你可能需要违背这个规则；但对于你能控制的代码，遵守它可以提高程序的可靠性、健壮性、可读性、可扩展性。

8.3 Item M34：如何在同一程序中混合使用 C++和 C

许多年来，你一直担心编制程序时一部分使用 **C++**一部分使用 **C**，就如同在全部用 **C** 编程的年代同时使用多个编译器来生成程序一样。没办法多编译器编程的，除非不同的编译器在与实现相关的特性（如 **int** 和 **double** 的字节大小，传参方式）上相同。但这个问题在语言的标准化中被忽略了，所以唯一的办法就是两个编译器的生产商承诺它们间兼容。**C++**和 **C** 混合编程时同样是这个问题，所以在实体混合编程前，确保你的 **C++**编译器和 **C** 编译器兼容。

确认兼容后，还有四个要考虑的问题：名变换，静态初始化，内存动态分配，数据结构兼容。

I 名变换

名变换，就是 C++编译器给程序的每个函数换一个独一无二的名字。在 C 中，这个过程是不需要的，因为没有函数重载，但几乎所有 C++程序都有函数重名（例如，流运行库就申明了几个版本的 operator<<和 operator>>）。重载不兼容于绝大部分链接程序，因为链接程序通常无法分辨同名的函数。名变换是对链接程序的妥协；链接程序通常坚持函数名必须独一无二。

如果只在 C++范围内，名变换不会影响你。如果你你有一个函数叫 drawline 而编译器将它变换为 xyzzy，你总使用名字 drawLine，不会注意到背后的 obj 文件引用的是 xyzzy 的。

如果 drawLine 位于 C 运行库中，那就是一个不同的故事了。你的 C++源文件包含的头文件中申明为：

```
void drawLine(int x1, int y1, int x2, int y2);
```

代码体中通常也是调用 drawLine。每个这样的调用都被编译器转换为调用名变换后的函数，所以写下的是

```
drawLine(a, b, c, d);          // call to unmangled function name
```

obj 文件中调用的是：

```
xyzzy(a, b, c, d);             // call to mangled function mame
```

但如果 drawLine 是一个 C 函数，obj 文件（或者是动态链接库之类的文件）中包含的编译后的 drawLine 函数仍然叫 drawLine；没有名变换动作。当你试图将 obj 文件链接为程序时，将得到一个错误，因为链接程序在寻找一个叫 xyzzy 的函数，而没有这样的函数存在。

要解决这个问题，你需要一种方法来告诉 C++编译器不要在这个函数上进行名变换。你不期望对用其它语言写的函数进行名变换，如 C、汇编、Fortran、LISP、Forth 或其它。（是的，这“其它”中应该包括 COBOL，但那时你将得到什么？（Yes, what-have-you would include COBOL, but then what would you have? ））总之，如果你调用一个名字为 drawLine 的 C 函数，它实际上就叫 drawLine，你的 obj 文件应该包含这样的一个引用，而不是引用进行了名变换的版本。

要禁止名变换，使用 C++的 extern 'C'指示：

```
// declare a function called drawLine; don't mangle
// its name
extern "C"
void drawLine(int x1, int y1, int x2, int y2);
```

不要以为有一个 extern 'C'，那么就应该同样有一个 extern 'Pascal'和 extern

'FORTRAN'。没有，至少在 C++标准中没有。不要将 extern 'C'看作是申明这个函数是用 C 语言写的，应该看作是申明在个函数应该被当作好象 C 写的一样而进行调用。（使用术语就是，extern 'C'意思是这个函数有 C 链接，但这个意思表达实在不怎么清晰。不管如何，它总意味着一件事：名变换被禁止了。）

例如，如果不幸到必须要用汇编写一个函数，你也可以申明它为 extern 'C'：

```
// this function is in assembler — don't mangle its name
extern "C" void twiddleBits(unsigned char bits);
```

你甚至可以在 C++函数上申明 extern 'C'。这在你用 C++写一个库给使用其它语言的客户使用时有用。通过禁止这些 C++函数的名变换，你的客户可以使用你选择的自然而直观的名字，而不用使用你的编译生成的变换后的名字：

```
// the following C++ function is designed for use outside
// C++ and should not have its name mangled
extern "C" void simulate(int iterations);
```

经常，你有一堆函数不想进行名变换，为每一个函数添加 extern 'C'是痛苦的。幸好，这没必要。extern 'C'可以对一组函数生效，只要将它们放入一对大括号中：

```
extern "C" {                          // disable name mangling for
                                      // all the following functions
  void drawLine(int x1, int y1, int x2, int y2);
  void twiddleBits(unsigned char bits);
  void simulate(int iterations);
  ...
}
```

这样使用 extern 'C'简化了维护那些必须同时供 C++和 C 使用的头文件的工作。当用 C++编译时，你应该加 extern 'C'，但用 C 编译时，不应该这样。通过只在 C++编译器下定义的宏__cplusplus，你可以将头文件组织得这样：

```
#ifdef __cplusplus
extern "C" {
#endif
  void drawLine(int x1, int y1, int x2, int y2);
  void twiddleBits(unsigned char bits);
  void simulate(int iterations);
  ...
#ifdef __cplusplus
}
```

```
#endif
```

顺便提一下，没有标准的名变换规则。不同的编译器可以随意使用不同的变换方式，而事实上不同的编译器也是这么做的。这是一件好事。如果所有的编译器使用同样的变换规则，你会误认为它们生成的代码是兼容的。现在，如果混合链接来自于不同编译器的 obj 文件，极可能得到应该链接错误，因为变换后的名字不匹配。这个错误暗示了，你可能还有其它兼容性问题，早些找到它比以后找到要好。

l 静态初始化

在掌握了名变换后，你需要面对一个 C++中事实：在 main 执行前和执行后都有大量代码被执行。尤其是，静态的类对象和定义在全局的、命名空间中的或文件体中的类对象的构造函数通常在 main 被执行前就被调用。这个过程称为静态初始化（参见 Item E47）。这和我们对 C++和 C 程序的通常认识相反，我们一直把 main 当作程序的入口。同样，通过静态初始化产生的对象也要在静态析构过程中调用其析构函数；这个过程通常发生在 main 结束运行之后。

为了解决 main()应该首先被调用，而对象又需要在 main()执行前被构造的两难问题，许多编译器在 main()的最开始处插入了一个特别的函数，由它来负责静态初始化。同样地，编译器在 main()结束处插入了一个函数来析构静态对象。产生的代码通常看起来象这样：

```
int main(int argc, char *argv[])
{
  performStaticInitialization();          // generated by the
                                          // implementation
  the statements you put in main go here;
  performStaticDestruction();             // generated by the
                                          // implementation
}
```

不要注重于这些名字。函数 performStaticInitialization() 和 performStaticDestruction()通常是更含糊的名字，甚至是内联函数（这时在你的 obj 文件中将找不到这些函数）。要点是：如果一个 C++编译器采用这种方法来初始化和析构静态对象，除非 main()是用 C++写的，这些对象将从没被初始化和析构。因为这种初始化和析构静态对象的方法是如此通用，只要程序的任意部分是 C++写的，你就应该用 C++写 main()函数。

有时看起来用 C 写 main()更有意义——比如程序的大部分是 C 的，C++部分只是一个支持库。然而，这个 C++库很可能含有静态对象（即使现在没有，以后可能会有——参见 Item M32），所以用 C++写 main()仍然是个好主意。这并不意味着你需要重写你的 C 代码。只要将 C 写的 main()改名为 realMain()，然后用 C++版本的 main()调用 realMain()：

```
extern "C"                                // implement this
```

```
int realMain(int argc, char *argv[]);     // function in C
int main(int argc, char *argv[])          // write this in C++
{
  return realMain(argc, argv);
}
```

这么做时，最好加上注释来解释原因。

如果不能用 C++写 main()，你就有麻烦了，因为没有其它办法确保静态对象的构造和析构函数被调用了。不是说没救了，只是处理起来比较麻烦一些。编译器生产商们知道这个问题，几乎全都提供了一个额外的体系来启动静态初始化和静态析构的过程。要知道你的编译器是怎么实现的，挖掘它的随机文档或联系生产商。

l 动态内存分配

现在提到动态内存分配。通行规则很简单：C++部分使用 new 和 delete（参见 Item M8），C 部分使用 malloc（或其变形）和 free。只要 new 分配的内存使用 delete 释放，malloc 分配的内存用 free 释放，那么就没问题。用 free 释放 new 分配的内存或用 delete 释放 malloc 分配的内存，其行为没有定义。那么，唯一要记住的就是：将你的 new 和 delete 与 mallco 和 free 进行严格的隔离。

说比做容易。看一下这个粗糙（但很方便）的 strdup 函数，它并不在 C 和 C++标准（运行库）中，却很常见：

```
char * strdup(const char *ps);          // return a copy of the
                                        // string pointed to by ps
```

要想没有内存泄漏，strdup 的调用着必须释放在 strdup()中分配的内存。但这内存这么释放？用 delete？用 free？如果你调用的 strdup 来自于 C 函数库中，那么是后者。如果它是用 C++写的，那么恐怕是前者。在调用 strdup 后所需要做的操作，在不同的操作系统下不同，在不同的编译器下也不同。要减少这种可移植性问题，尽可能避免调用那些既不在标准运行库中（参见 Item E49 和 Item M35）也没有固定形式（在大多数计算机平台下）的函数。

l 数据结构的兼容性

最后一个问题是在 C++和 C 之间传递数据。不可能让 C 的函数了解 C++的特性的，它们的交互必须限定在 C 可表示的概念上。因此，很清楚，没有可移植的方法来传递对象或传递指向成员函数的指针给 C 写的函数。但是，C 了解普通指针，所以想让你的 C++和 C 编译器生产兼容的输出，两种语言间的函数可以安全地交换指向对象的指针和指向非成员的函数或静态成员函数的指针。自然地，结构和内建类型（如 int、char 等）的变量也可自由通过。

因为 C++中的 struct 的规则兼容了 C 中的规则，假设“在两类编译器下定义的同一结构将按同样的方式进行处理”是安全的。这样的结构可以在 C++和 C 见安全地来回传递。如

果你在 C++版本中增加了非虚函数，其内存结构没有改变，所以，只有非虚函数的结构（或类）的对象兼容于它们在 C 中的孪生版本（其定义只是去掉了这些成员函数的申明）。增加虚函数将结束游戏，因为其对象将使用一个不同的内存结构（参见 Item M24）。从其它结构（或类）进行继承的结构，通常也改变其内存结构，所以有基类的结构也不能与 C 函数交互。

就数据结构而言，结论是：在 C++和 C 之间这样相互传递数据结构是安全的——在 C++和 C 下提供同样的定义来进行编译。在 C++版本中增加非虚成员函数或许不影响兼容性，但几乎其它的改变都将影响兼容。

! 总结

如果想在同一程序下混合 C++与 C 编程，记住下面的指导原则：

- ! 确保 C++和 C 编译器产生兼容的 obj 文件。
- ! 将在两种语言下都使用的函数申明为 extern 'C'。
- ! 只要可能，用 C++写 main()。
- ! 总用 delete 释放 new 分配的内存；总用 free 释放 malloc 分配的内存。
- ! 将在两种语言间传递的东西限制在用 C 编译的数据结构的范围内；这些结构的 C++版本可以包含非虚成员函数。

8.4 Item M35：让自己习惯使用标准 C++语言

自 1990 年出版以来，《The Annotated C++ Reference Manual 》（见原书 P285，附录：推荐读物）是最权威的参考手册供程序员们判断什么是 C++拥有的而什么不是。在它出版后的这些年来，C++的 ISO/ANSI 标准已经发生了大大小小的变化了（主要是扩充）。作为权威手册，它已经不适用了。

在《ARM》之后 C++发生的变化深远地影响了写出的程序的优良程度。因此，对程序员们来说，熟悉 C++标准申明的与《ARM》上描述的关键性不同是非常重要的。

ISO/ANSI C++标准是厂商实现编译器时将要考虑的，是作者们准备出书时将要分析的，是程序员们在对 C++发生疑问时用来寻找权威答案的。在《ARM》之后发生的最主要变化是以下内容：

- ! 增加了新的特性：RTTI、命名空间、bool，关键字 mutable 和 explicit，对枚举的重载操作，已及在类的定义中初始化 const static 成员变量。
- ! 模板被扩展了：现在允许了成员模板，增加了强迫模板实例化的语法，模板函数允许无类型参数，模板类可以将它们自己作为模板参数。
- ! 异常处理被细化了：异常规格申明在编译期被进行更严格的检查，unexpected()函数现在可以抛一个 bad_exception 对象了。
- ! 内存分配函数被改良了：增加了 operator new[]和 operator delete[]函数，operator new/new[]在内存分配失败时将抛出一个异常，并有一个返回为 0（不抛

异常）的版本供选择。（见 **Effective C++ Item 7**）

- 增加了新的类型转换形式：**static_cast**、**dynamic_cast**、**const_cast**，和 **reinterpret_cast**。
- 语言规则进行了重定义：重定义一个虚函数时，其返回值不需要完全的匹配了（如果原来返回基类对象或指针或引用，派生类可以返回派生类的对象、指针或引用），临时对象的生存期进行了精确地定义。

绝大部分变化都描述于《**The Design and Evolution of C++** 》（见原书 **P285**，附录：推荐读物）。现在的 **C++**教科书（那些写于 **1994** 年以后的）应该也都包含了它们。（如果发现哪本没有，那么扔掉它。）另外，本书包含了一些例子来展示任何使用这些新特性。欲知祥情，请看索引表。

C++的这些变化在标准运行库的变化面前将黯然失色。此外，标准运行库的演变从没象语言本身这样被宣扬过。例如，《**The Design and Evolution of C++**》几乎没有提及标准运行库。讨论运行库的书籍都有些过时，因为运行库在 **1994** 年后发生了非常巨大的变化。

标准运行库的功能分为下列类别（参见 **Effective C++ Item 49**）**:**

- 支持标准 **C** 运行库。不用担心，**C++**仍然记得它的根源。进行了一些细微的调整，使得 **C++**版本的 **C** 运行库顺应了 **C++**更严格的类型检查，但其功能和效果，在 **C** 运行库是怎么样的，在 **C++**中还是怎么样。
- 支持 **string** 类型。标准 **C++**运行库实现组的领衔人物 **Mike Vilot** 被告知“如果没有一个标准的 **string** 类型，那么将血流于街!”（有些人是如此激动。）平静一下，把板斧和棍子放下——标准 **C++**运行库有了 **string** 类型。
- 支持本地化。不同的文化有不同的字符集，以及在显示日期和时间、排序字符串、输出货币值……等等时有不同的习惯。标准运行库对本地化的支持便于开发同时供不同文化地区使用的程序。
- 支持 **I/O** 操作。流运行库仍然有部分保留在 **C++**标准中，但标准委员会作了小量的修补。虽然部分类被去除了（特别是 **iostream** 和 **fstream**），部分类被替换了（例如，以 **string** 为基础的 **stringstream** 类替换了以 **char***为基础的 **strstream** 类，现在已不提倡使用 **strstream** 类了），但标准流类的基本功能含概了以前的实现的基本功能。
- 支持数学运算。复数，**C++**教课书必谈的东西，最终纳入了标准运行库。另外，运行库包含了特别的数组类（**valarray**）以免混淆。这些数组类比内建类型的数组有更好的性能，尤其在多 **CPU** 系统下。运行库也提供了部分常用的运算函数如加法和减法。
- 支持通用容器和运算。标准 **C++**运行库带了一组模板类和模板函数，称为标准模板库（**STL**）。**STL** 是标准 **C++**运行库中最革命性的部分。我将在下面概述它的特性。

在介绍 **STL** 前，必须先知道标准 **C++**运行库的两个特性。

第一，在运行库中的几乎任何东西都是模板。在本书中，我谈到过运行库中的 **string** 类，实际上没有这样的类。其实，有一个模板类叫 **basic_string** 来描述字符序列，它接受一个字符类型的参数来构造此序列，这使得它能表示 **char** 串、**wide char** 串、**Unicode char** 串等等。

我们通常认为的 **string** 类是从 **basic_string<char>**实例化而成的。用于它被用得如此广泛，标准运行库作了一个类型定义：

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

这其实仍然隐藏了很多细节，因为 **basic_string** 模板带三个参数；除了第一个外都有默认参数。要全面理解 **string** 类型，必须面对这个未经删节的 **basic_string**：

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

使用 **string** 类型并不需要了解这个官方文章，因为类型定义使得它表现得如同不是模板类。无论你需要定制存入 **string** 的字符集，或调整这些字符的行为，或控制分配给 **string** 的内存，这个 **basic_string** 模板都能让你完成这些事。

设计 **string** 类型时采用的逼近法——归纳其行为然后推广为模板——被标准 **C++**运行库广泛采用。**IOstream**？它们是模板；一个类型参数决定了这个流的特征。复数？也是模板；一个类型参数决定了数字被怎么样存储。**Valarray**？模板；一个类型参数表明了数组里面存了什么。如果你不熟悉模板，现在正是个熟悉的好机会。

另外需要知道的是：标准运行库将几乎所有内容都包含在命名空间 **std** 中。要想使用标准运行库里面的东西而无需特别指明运行库的名称，你可以使用 **using** 指示或使用（更方便的）**using** 申明（参见 **Effective C++ Item 28**）。幸运的是，这种重复工作在你**#include** 恰当的头文件时自动进行。

Ⅰ 标准模板库

标准 **C++**运行库中最大的新闻就是 **STL**，标准模板库。（因为 **C++**运行库中的几乎所有东西都是模板，这个 **STL** 就不怎么贴切了。不过，运行库中包容器和算法的部分，其名字不管好还是坏，反正现在就叫了这个。）

STL 很可能影响很多——恐怕是绝大多数——**C++**运行库的结构，所以熟悉它的基本原理是很重要的。它们并不难理解。**STL** 基于三个基本概念：包容器（**container**）、选择子（**iterator**）和算法（**algorithms**）。包容器是被包容对象的封装；选择子是类指针的对象让你能如同使用指针操作内建类型的数组一样操作 **STL** 的包容器；算法是对包容器进行处理的函数，并使用选择子来实现的。

联想一下 C++（和 C）中数组的规则，就太容易理解 STL 的想法了。真的只需要明确一点：一个指向数组的指针可以正确地指出数组的任意元素或刚刚超出数组范围的那个元素。如果指向了那个超范围的元素，它将只能与其它指向此数组的指针进行地址比较；对其进行反引用，其结果为未定义。

我们可以利用这条规则来实现在数组中查找一个特定值的函数。对一个整型数组，函数可能是这样的：

```
int * find(int *begin, int *end, int value)
{
  while (begin != end && *begin != value) ++begin;
  return begin;
}
```

这个函数在 begin 与 end 之间（不包括 end——end 指向刚刚超出数组范围的元素）查找 value，返回第一个值为 value 的元素；如果没有找到，它返回 end。

返回 end 来表示没找到，看起来有些可笑。返回 0（NULL 指针）不是更好吗？确实 NULL 看起来更自然，但并不意味着更“好”。find()函数必须返回特别的指针值来表明查找失败，就此目的而言，end 指针与 NULL 指针效果相同。但，如我们将要看到的，end 指针在推广到其它包容器类型时比 NULL 指针好。

老实说，这可能不是你实现 find()函数的方法，但它不是不切实际的，它的推广性很好。看完下面的例子，你将掌握 STL 的思想。

你可以这么使用 find()函数：

```
int values[50];
...
int *firstFive = find(values,        // search the range
                      values+50,     // values[0] - values[49]
                      5);            // for the value 5
if (firstFive != values+50) {        // did the search succeed?
  ...                                // yes
}
else {
  ...                                // no, the search failed
}
```

你也可以只搜索数组的一部分：

```
int *firstFive = find(values,        // search the range
                      values+10,     // values[0] - values[9]
```

```
                    5);          // for the value 5
int age = 36;
...
int *firstValue = find(values+10,    // search the range
                       values+20,    // values[10] - values[19]
                       age);         // for the value in age
```

find()函数内部并没有限制它只能对 **int** 型数组操作，所以它可以实际上是一个模板：

```
template<class T>
T * find(T *begin, T *end, const T& value)
{
  while (begin != end && *begin != value) ++begin;
  return begin;
}
```

在转为模板时，注意我们由“传值”改为了“传 **const** 型引用”。因为现在可以处理任意类型了，我们不得不考虑传值的代价了。在每次调用过程中，每个传值的参数都要有构造函数和析构函数的开销。通过传引用（它不需要构造和析构任何对象），我们避免了这个开销（参见 **Effective C++ Item 22**）。

这个模板很漂亮，但还可以推广得更远些。注意在 **begin** 和 **end** 上的操作。只有“不等于”比较、反引用、前缀自增（参见 **More Effective C++ Item M6**）、拷贝（对函数返回值进行的，参见 **More Effective C++ Item M19**）。这就是我们涉及到的全部操作，所以为什么限制 **find()**只能使用指针呢？为什么不允许其它支持这些操作的对象呢？这样一来就可以将 **find()**从内嵌类型的指针中解放出来。例如，我们可以为一个链表定义一个类指针对象，其前缀自增操作将使自己指向链表的下一个元素。

这就是 **STL** 的选择子的概念。选择子就是被设计为操作 **STL** 的包容器的类指针对象。它们是 **Item M28** 中讲的灵巧指针的堂兄，只是灵巧指针的功能更强大。但从技术角度看，它们的实现使用了同样的技巧。

有了作为类指针对象的选择子的概念，我们可以用选择子代替 **find()**中的指针。改写后的 **find()**类似于：

```
template<class Iterator, class T>
Iterator find(Iterator begin, Iterator end, const T& value)
{
  while (begin != end && *begin != value) ++begin;
  return begin;
}
```

恭喜！你恰巧写出了标准模板库的一部分。STL 中包含了很多使用包容器和选择子的算法，find()是其中之一。

STL 中的包容器有 bitset、vector、list、deque、queue、priority-queue、stack、set 和 map，你可以在其中任一类型上使用 find()，例如：

```
list<char> charList;                    // create STL list object
                                        // for holding chars
...
// find the first occurrence of 'x' in charList
list<char>::iterator it = find(charList.begin(),
                               charList.end(),
                               'x');
```

“嗨！”，我听到你大叫，“这和前面使用数组的例子看起来一点都不象！”哈，但它是一样的；你只是必须知道 find()期望什么输入。

要对 list 对象调用 find()，你必须提供一个指向 list 中的第一个元素的选择子和一个越过 list 中最后一个元素的选择子。如果 list 类不提供帮助，这将有些难，因为你无法知道 list 是怎么实现的。幸好，list（和其它所有 STL 的包容器一样）被责令提供成员函数 begin()和 end()。这些成员函数返回你所需要的选择子，供你传给 find()用。

当 find()执行完成时，它返回一个选择子对象指向找到的元素（如果元素有的话）或 charList.end()（如果没有的话）。因为你不知道 list 是怎么实现的，所以也无法知道 list 内部的选择子是怎么实现的。于是，你怎么知道 find()返回什么类型的对象？如其它所有 STL 的包容器一样，list 再次提供了帮助：它提供了一个类型重定义，iterator 就是 list 内部使用的选择子的类型。既然 charList 是一个包容 char 的 list，它内部的选择子类型就是 list<char>::iterator（也就是上面的例子中使用的）。（每个 STL 包容器类实际上定义了两个选择子类型，iterator 和 const_iterator。前一个象普通指针，后一个象指向 const 对象的指针。）

同样的方法也完全适用于其它 STL 包容器。此外，C++指针也是 STL 选择子，所以，最初的数组的例子也能适用 STL 的 find()函数：

```
int values[50];
...
int *firstFive =   find(values, values+50, 5);     // fine, calls
                                                   // STL find
```

STL 其实非常简单。它只是收集了遵从同样规则的类模板和函数模板。STL 的类提供如同 begin()和 end()这样的函数，这些函数返回类型定义在此类内的选择子对象。STL 的算法函数使用选择子对象操作这些类。STL 的选择子类似于指针。这就是 STL 的全部。它没有

庞大的继承层次，也没有虚函数。只是影响类模板和函数模板，及它们所遵守的规则。

这导致了另外一个发现：**STL** 是可扩充的。你可以在 **STL** 家族中加入自己的包容器，算法和选择子。总要你遵守 **STL** 的规则，**STL** 中的包容器将可以使用你的算法，你的包容器上也可使用 **STL** 的算法。当然，你的模板不会成为标准 **C++**运行库的一部分，但它们将用同样的规则来编译，并可被重用。

C++运行库中还有更多我没有写出来的东西。在你能高效使用运行库前，你必须跳出我在这儿画的框去学更多的东西；在你能写自己的 **STL** 兼容模板前，你必须掌握更多的 **STL** 规则。标准 **C++**运行库比 **C** 运行库丰富得太多了，花时间来熟悉它是值得的（参见 **Item E49**）。此外，掌握设计运行库的原则（通用、可扩充、可定制、高效、可复用）也是值得的。通过学习标准 **C++**运行库，你不光涨了知识知道有什么现成的可用在自己的软件中，你也将学到如何更高效地使用 **C++**的特性，也能掌握如何设计更好的代码库。

9. 附录

9.1 推荐读物

可能你对 **C++** 相关资讯的欲望还未饱足。别担心，还有更多— 多得是。以下列出我对 **C++** 进阶读物的推荐。我想不必特别声明，推荐当然是主观的。不过我愿意再说一次：推荐是主观的，你有选择。

I 书籍

C++ 书籍成百成千，新的竞争者以极大的频率加入这场骚动之中。我并未看过所有这些书籍，仔细阅读过的也不是非常多，但我的经验是，其中有的非常好，有的并不理想。

下面开出来的书单是我自己在 **C++** 软体开发过程中遇到问题时的谘询对象。其他好书当然也有，但这些是我自己使用过的，所以我可以放心推荐。

从描述「语言本身」的书籍开始，应该是个好起点。除非你非常在乎这些书籍与官方标准文件之间的一些细微差异，否则我建议你阅读它们：

- **The Annotated C++ Reference Manual, Margaret A. Ellis andBjarne Stroustrup,Addison-Wesley, 1990, ISBN 0-201-51459-1.**
- **The Design and Evolution of C++, Bjarne Stroustrup, Addison-Wesley, 1994,ISBN 0-201-54330-3.**

这两本书涵盖的不只是语言本身的描述，也解释了隐藏在设计之下的基本原理—。那是你无法从官方标准文件中获得的东西。**The Annotated C++ Reference Manual** 如今已经不够完整（它出版之後，**C++** 又新增了一些语言特性— 见条款 **35**），从某方面说已经过气，但它仍然是语言核心方面（包括 **templates** 和 **exceptions**）的最佳参考书籍。**The Annotated C++ Reference Manual** 遗漏的主题，大部份已涵盖於 **The Design and Evolution of C++** 之

中，唯独欠缺 **Standard Template Library**（见条款 **35**）的讨论。这些书籍都不是学习指南或自修书，它们是参考书籍，但如果你不了解这些书籍所谈的东西，实在不能说是真正了解 **C++**。

至於 **C++** 语言及标准程式库方面的一般性参考书籍，没有谁比得上 **C++** 语言创造者所写的书：

- **The C++ Programming Language (Third Edition), BjarneStroustrup, Addison-Wesley, 1997, ISBN 0-201-88954-4.**

从 **C++** 语言诞生伊始，**Stroustrup** 便涉及其设计、实作、应用、以及标准化。他知道的恐怕比任何其他人都多。他对 **C++** 语言特性的描述，形成了这本密度极高的读物。密度高主要是因为，资讯就是那么多。书中与 **C++** 标准程式库相关的各章篇幅，提供了 **C++** 重要部份的一个良好导入。

如果你即将跨越语言本身，努力思考如何有效运用 **C++**，可以考虑我的另一本书：

- **Effective C++, Second Edition: 50 Specific Ways to Improve YourPrograms andDesigns, Scott Meyers, Addison-Wesley, 1998, ISBN0-201-92488-9.**

此书组织风格类似本书，但涵盖不同（可说是较为基础）的题材。

有一本书的基调和我的 **Effective C++** 差不多，但涵盖主题不同：

- **C++ Strategies and Tactics, Robert Murray, Addison-Wesley, 1993,ISBN 0-201-56382-7.**

Murray 的书在 **template** 相关设计方面特别著重，他在这上面贡献了两章。另有一章专注於「从 **C** 的开发迁徙至 **C++** 的开发」这个重要题目。我对 **reference counting**（参用计数，条款 **29**）的讨论亦是以 **C++ Strategies and Tactics** 的观念为基础。

如果你是那种喜欢从读码过程中学习程式技术的人，下面这本书适合你：

- **C++ Programming Style, Tom Cargill, Addison-Wesley, 1992,ISBN 0-201-56365-7.**

此书的每一章，一开始都以某些公开发行的 **C++** 软体做为例子，示范如何正确进行某些事情。然後 **Cargill** 开始解剖— 活体解剖— 每个程式，找出可能出现麻烦的地点、不良的设计选择，短暂易碎的实作决定、以及根本就错误的动作。然後他重写每一个例子，消除那些缺点。经过他的改造，程式码更强固、更易维护、更有效率、亦更容易移植，并且仍能解决原先的问题。任何 **C++** 程式员，特别是对那些需要检阅别人程式的人，最好能够留心此书带给我们的课题。

C++ Programming Style 没有讨论的一个主题是 **exceptions**。**Cargill** 把他那对语言特性有特异功能的鹰眼摆在以下文章中，该文告诉我们，为什么写出「面对 **exception** 依然安全（所谓 **exception-safe**）」的程式，其困难度比大部份程式员所了解的更困难：

"Exception Handling: A False Sense of Security," C++ Report,Volume 6, Number9,

November-December 1994, pages 21-24.

如果你重视 exceptions 的使用，动手之前请先读过上篇文章。

一旦你精通了 C++ 基本面，准备开始收成，你必须让自己熟悉：

- Advanced C++: Programming Styles and Idioms, James Coplien,Addison-Wesley,1992, ISBN 0-201-54855-0.

我常称此书为"the LSD book"（译注：Lysergic Acid Diethylamide 是一种迷幻药），因为它有紫色的封面，而且它会使你的心神膨胀。Coplien 涵盖某些直接而明确的素材，但他真正的焦点在於告诉你如何在 C++ 中做到你不以为能够做到的事情。

想要将物件建构於另一个物件之上吗？他告诉你怎么做。想要回避强烈型别检验（strong typing）吗？他给你一个方法。想要在程式执行时为 classes 加上资料和函式吗？他为你解释如何进行。大部份时候，你大概只是循著他所叙述的技术前进，尽情地浏览，但有时候这些技术刚好可以提供你手上某个棘手问题的解答。

犹有进者，它揭露了 C++ 可以做到什么样的事情。这本书可能令你骇怕，可能使你茫然，但是当你读完它，你再也不会像以前一样地看待 C++ 了。

如果你打算设计和实作 C++ 程式库，漏看以下书籍，只能说是有勇无谋：

- Designing and Coding Reusable C++, Martin D. Carroll andMargaret A. Ellis,Addison-Wesley, 1995, ISBN 0-201-51284-X.

Carroll 和 Ellis 讨论了程式库设计和实作上的许多实用方向，这些方向常被每个人忽略。好的程式库的特色是小、快、可扩充、容易升级、在 template 具现化时有优雅的表现、威力强大、而且稳健。没有一个人能够达到上述每一项要求，所以你必须有所取舍。Designing and Coding Reusable C++ 验证这些取舍，并提供现实的忠告，告诉你如何达成你所选择的目标。

不论你写的软体是否应用於科学或工程，你都应该看看这本书：

- Scientific and Engineering C++, John J. Barton and Lee R.Nackman, Addison-Wesley, 1994, ISBN 0-201-53393-6.

此书第一部份为 FORTRAN 程式员解释 C++（这可不是件值得羡慕的工作），但是稍後所涵盖的技术几乎适用於任何领域。书中对 templates 的广泛运用，简直像一场革命；这或许是目前为止最先进的应用，我猜一旦你看过这些作者以 templates 完成的奇迹，你再也不会以为所谓 templates 只不过是比 macros 好一点的玩意儿。

压轴的是物件导向软体开发过程的 patterns 训练（见 p123）。这个主题描述於：

- Design Patterns: Elements of Reusable Object-Oriented Software,Erich Gamma,Richard Helm, Ralph Johnson, and John Vlissides,Addison-Wesley, 1995, ISBN 0-201-63361-2.

这本书对於 patterns 背後的概念，提供了一份导览。其主要贡献在於对可应用於众多

领域之 23 个基本 patterns，加以分类整理。任何人翻阅这份分类目录，几乎一定可以找到一个你曾经於过去某时候自行发明的 pattern。而同时你也几乎一定会发现此书的设计优於你的设计。书中所提的 patterns 名称，几乎已经成为物件导向设计领域的标准辞汇。如果不知道这些名称，可能会造成你和你的同事之间沟通上的危机。此书特别重视软体应该如何设计和实作，才能将未来的演化优雅地纳入（见条款 32 和 33）。

Design Patterns 也以 CD-ROM 的形式呈现：

Design Patterns CD: Elements of Reusable Object-OrientedSoftware, Erich Gamma,Richard Helm, Ralph Johnson, and JohnVlissides, Addison-Wesley, 1998, ISBN 0-201-63498-8.

l 杂志刊物

喜欢 C++ 硬梆梆素材的家伙们，这本刊物大概是你生活中唯一的乐趣：

l C++ Report, SIGS Publications, New York, NY.

这本刊物做了一个神志清醒的决定，放弃"C++ only" 的老根。不过由於持续增加与特定领域和特定系统有关的程式设计主题，并且做得很好，所以即使 C++ 方面的题材偶尔「程度不够深」，整体而言仍然很有价值。

如果你喜欢 C 甚於 C++，或是如果你发现 C++ Report 的题材太极端，或许可以在这本刊物中找到一些符合自己口味的文章：

l C/C++ Users Journal, Miller Freeman, Inc., Lawrence, KS.

如其名称所示，这本刊物兼含 C 和 C++。其中 C++ 文章对读者程度的要求，比 C++ Report 所要求的低。此外，编辑群对於作者的约束很紧，所以刊出的文章相对比较主流。这有助於过滤掉狂暴边缘的某些想法，不过这也限制了你真正看到「锐利」技术的机会。

l 网路社群（Usenet Newsgroups）

有三个 Usenet 讨论群专注於 C++ 这个大主题。一般性的、什么都可以上去的讨论群是 comp.lang.c++。这个场所进行全方位交易，从高阶程式技术的细微讨论，到胡说八道似的瞎扰和（谁谁谁喜欢 C++ 啦，谁谁谁讨厌 C++ 啦），再到全世界大学生在此火烧屁股地寻求作业协助，都有！这个讨论群上的卷籍实在是太多了。除非你有数个小时的自由时间，我想你会运用过滤器来帮你筛出粗糠中的小麦。选一个好点儿的过滤软体吧，粗糠很多呢！

1995 年 11 月， 一个有主持人驻守的 comp.lang.c++ 诞生了。这个名为 comp.lang.c++.moderated 的讨论群，也是用於 C++ 及其相关主题的一般讨论，但主持人会删除特定平台上的问题和见解、线上常见问答集（on-line FAQ，推荐读物 Frequently Asked Questions）已涵盖的问题、口水战（不管是为了什么）、以及大部份 C++ 程式开发者比较不感兴趣的话题。

一个范围更窄的讨论群是 comp.std.c++，它专注於 C++ 标准规格的讨论。语言方面的专家多出没於此讨论群中。如果你那吹毛求疵的 C++ 问题在其他讨论群上一直没有人回答，

或回答得不令你满意，这里是发问的好地方。这个讨论群有主持人，所以「良讯/杂讯」比相当令人满意；你不会在这里看到任何作业请托。

9.2 一个 auto_ptr 的实现实例

Items M9、**M10**、**E26**、**E31** 和 **E32** 证明了 **auto_ptr** 模板类的非同寻常的作用。不幸的是，目前很少有编译器地提供了一个“正确”的实现（注 **1**）。**Items M9** 和 **M28** 大致描述了你怎么自己实现一个，但从事实际项目时有一个更详尽的版本就太好了。

下面是两个 **auot_ptr** 的实现。第一个版本文档化了类的接口并在类的定义体外面实现了所有的成员函数。第二个版本将所有的成员函数都实现在定义体内了。在文体上，第二个实现不如第一个，因为它没有将类的接口从实现中分离出来。但，**auto_ptr** 只是一个简单的类，所以第二个实现比第一个清晰得多。

这是有专门接口申明的 **auto_ptr** 模板：

```
template<class T>
class auto_ptr {
public:
  explicit auto_ptr(T *p = 0);              // Item M5 有“explicitfor”
                                            // 的描述
  template<class U>                         // 拷贝构造函数成员模板
  auto_ptr(auto_ptr<U>& rhs);               //  （见 Item M28）：
                                            // 用另一个类型兼容的
                                            // auto_ptr 对象
                                            // 初始化一个新的 auto_ptr 对象
  ~auto_ptr();
  template<class U>                         // 赋值操作成员模板
  auto_ptr<T>&                              // （见 Item M28）：
  operator=(auto_ptr<U>& rhs);              // 用另一个类型兼容的
                                            // auto_ptr 对象给它赋值
  T& operator*() const;                     // 见 Item M28
  T* operator->() const;                    // 见 Item M28
  T* get() const;                           // 返回包容指针的
                                            // 当前值
  T* release();                             // 放弃包容指针的
                                            // 所有权，
                                            // 并返回其当前值
```

```
  void reset(T *p = 0);                    // 删除包容指针，
                                           // 获得指针 p 的所有权
private:
  T *pointee;
template<class U>                          // 让所有的 auto_ptr 类
friend class auto_ptr<U>;                  // 成为友元
};
template<class T>
inline auto_ptr<T>::auto_ptr(T *p)
: pointee(p)
{}
template<class T>
  inline auto_ptr<T>::auto_ptr(auto_ptr<U>& rhs)
  : pointee(rhs.release())
  {}
template<class T>
inline auto_ptr<T>::~auto_ptr()
{ delete pointee; }
template<class T>
  template<class U>
  inline auto_ptr<T>& auto_ptr<T>::operator=(auto_ptr<U>& rhs)
  {
    if (this != &rhs) reset(rhs.release());
    return *this;
  }
template<class T>
inline T& auto_ptr<T>::operator*() const
{ return *pointee; }
template<class T>
inline T* auto_ptr<T>::operator->() const
{ return pointee; }
template<class T>
inline T* auto_ptr<T>::get() const
{ return pointee; }
```

```
template<class T>
inline T* auto_ptr<T>::release()
{
  T *oldPointee = pointee;
  pointee = 0;
  return oldPointee;
}
template<class T>
inline void auto_ptr<T>::reset(T *p)
{
  if (pointee != p) {
    delete pointee;
    pointee = p;
  }
}
```

这是所有函数定义在类定义体内的 **auto_ptr** 模板。如你所见，它不会搞乱大脑：

```
template<class T>
class auto_ptr {
public:
  explicit auto_ptr(T *p = 0): pointee(p) {}
  template<class U>
  auto_ptr(auto_ptr<U>& rhs): pointee(rhs.release()) {}
  ~auto_ptr() { delete pointee; }
  template<class U>
  auto_ptr<T>& operator=(auto_ptr<U>& rhs)
  {
    if (this != &rhs) reset(rhs.release());
    return *this;
  }
  T& operator*() const { return *pointee; }
  T* operator->() const { return pointee; }
  T* get() const { return pointee; }
  T* release()
  {
```

```
    T *oldPointee = pointee;
    pointee = 0;
    return oldPointee;
  }
  void reset(T *p = 0)
  {
    if (pointee != p) {
      delete pointee;
      pointee = p;
    }
  }
  private:
    T *pointee;
  template<class U> friend class auto_ptr<U>;
  };
```

如果你所用的编译器还不支持“**explicit**”，可以安全地用**#define** 取消它的存在：

```
#define explicit
```

这不会造成 **auto_ptr** 的任何功能减弱，但导致轻微的安全性减弱。详见 **Item M5**。

如果你的编译器不支持成员模板，你可以使用非模板的 **auto_ptr** 拷贝构造函数和赋值操作（描述在 **Item M28**）。这将造成你的 **auto_ptr** 在使用上有些小小的不方便，但没有其它方法能模仿成员模板的行为，唉！如果成员模板（或语言中的其它一些特性）对你非常重要，告诉你的编译器提供商。越多的用户要求新的语言特性，提供商将越快地实现它们。

l 注 1：

这主要是因为 **auto_ptr** 的标准长年来一直没有确定。其最终描述被采纳于 **1997** 年 **11** 月。其细节可参考本书的主页。注意，此处的 **auto_ptr** 版本在实现上比正式版本在具体细节上有小小的省略：实际上 **auto_ptr** 位于名字空间 **std** 中（见 **Item M35**）并且其成员函数承诺不抛任何异常。

9.3 在 C++ 中计算物件个数（Objects Counting in C++）译者：陈崴

在 **C++** 中，对某个 **class** 所产生出来的 **objects** 保持正确的计数，是办得到的，除非你面对一些疯狂份子。

侯捷注：本文系北京《程序员》杂志 **2001/08** 的文章。译笔顺畅，技术饱满。承译者陈崴先生与《程序员》杂志负责人蒋涛先生答允，转载於此，以飨台湾读者，非常感谢。未得陈崴先生与蒋涛先生二人之同意，任何人请勿将此文再做转载。

译注：本文发表日期比 ***C++ Standard*** 发表日期早，但文中内容皆符合 ***C++ Standard***。译文保留原作时态，并未修改。以下是译文所采用的几个特别请读者注意的术语：

client：　客端。
type：　型别。为避免和其他近似术语混淆，本文译为「型别」而非「类型」。
instantiated：具现化。「针对一个 **template**，具体实现出一份实体」的意思。
instance：　实体。「案例」是绝对错误的译法。
parameter：　叁数。或称型式叁数，形叁。
argument：　引数。或称实质叁数，实叁。

至於 **class, object, data member, member function, constructor, destructor, template** 等术语，皆保留不译。

有时候，容易的事情虽然容易，但它们还是隐藏着某种微妙。举个例子，假设你有个 **class** 名为 **Widget**，你希望有某种办法找出程式执行期间究竟存在着多少个 **Widget objects**。方法之一（不但容易实作而且答案正确）就是为 **Widget** 准备一个 **static** 计数器，每当 **Widget constructor** 被呼叫，就将该计数器加一，每当 **Widget destructor** 被呼叫，就将该计数器减一。此外你还需要一个 **static** 成员函式 **howMany( )**，用来回报目前存在多少个 **Widget objects**。如果 **Widget** 什麼都没做，单单只是追踪其 **objects** 个数，那麼它看起来大约像这样：

```
class Widget {
public:
    Widget() { ++count; }
    Widget(const Widget&) { ++count; }
    ~Widget() { --count; }

    static size_t howMany()
    { return count; }

private:
    static size_t count;
};

// count 的定义。这应该放在一个实作档中。
size_t Widget::count = 0;
```

这可以有效运作。可别忘了实作出 **copy constructor**，因为编译器自动为 **Widget** 产生的那个 **copy constructor** 不会知道将 **count** 加一。

如果你只需要为 **Widget** 做计数工作，你已经完成了你的任务。但有时候你得为许多 **classes** 做相同的计数工作。一再进行重复的工作会令人沉闷而生厌，而沉闷与厌恶感会导致错误的发生。为了阻止这种局面，最好能够将上述的「物件计数」（**object-counting**）程式码包装起来，使它能够被任何 **class** 重复运用。理想的包装应该符合以下条件：

- 容易使用 — 让那些需要此等服务的 **class** 设计者只需最小量的工作。最理想的情况是，他们不必做任何事情，只需宣称『我要对这种型别的 **objects** 进行计量』就好。
- 有效率 — 使用者不需被课徵任何非必要的空间税或时间税。
- 绝对安全 — 不可能突然导致一个错误计量。（我们不打算理会那些蓄意破坏的恶意使用者，他们会刻意试着混淆计量。在 **C++** 中，诸如此类的使用者总是能找到办法来满足他们卑鄙而居心不良的行为）

暂停一下，想一想，你如何实作出一个可复用的「物件计数」套件，并满足以上所有目标。这或许比你所预想的还要困难。如果它真的如你所想像的那麼容易，你就不会在这本刊物上看到这篇文章了。

I new, delete, 和 Exceptions

虽然你正全心全意地解决「物件计数」相关问题，请允许我把焦点暂时切换到一个似乎无关的主题上。这个主题就是「当 constructors 丢出异常，new 和 delete 之间的关系」。当你要求 C++ 动态配置一个 object，你会像这样地使用 new 运算式：

```
class ABCD { ... }; // ABCD = "A Big Complex Datatype"
ABCD *p = new ABCD; // 这是一个 new 运算式
```

这个 new 运算式的意义在语言层面已经确定，其行为无论如何无法被改变。它做两件事情。第一，它呼叫一个名为 operator new 的记忆体配置函式。这个函式的责任是找出足够置放一个 ABCD object 的记忆体。如果配置成功，new 运算式接下来就唤起一个 ABCD constructor，在 operator new 找出的那块空间上建立一个 ABCD object。

但是，假设 operator new 丢出一个 std::bad_alloc 异常，会怎样？这种异常表示，动态配置记忆体的任务失败了。在上述的 new 运算式中，有两个函式可能发出这样的异常。第一个是 operator new，它企图找出足够的记忆体来放置一个 ABCD object。第二个是接下来执行的 ABCD constructor，它企图把生鲜记忆体转换为一个有效的 ABCD object。

如果异常来自 operator new，表示没有任何记忆体被配置出来。然而如果 operator new 成功而 ABCD constructor 发出异常，很重要的一件事就是将 operator new 所配置的记忆体释放掉。如果不这样，程式就会发生记忆体遗失（memory leak）。客端（也就是要求产生一个 ABCD object 的那段程式码）不可能知道到底哪一个函式发出异常。

多年以来，这是 C++ 标准草案中的一个漏洞，1995 年三月，C++ 标准委员会采纳了一个提议：如果在 new 运算式动作期间，operator new 配置记忆体成功而後继的 constructor 丢出异常，执行期系统（runtime system）就必须自动释放被 operator new 配置出来的记忆体。这个释放动作由 operator delete 执行，此函式类似 operator new。（详见本文最後方块栏目内的「placement new 和 placement delete」。）

这个「new 运算式和 operator delete 之间的关系」，会在我们企图将「物件计数机制」自动化的过程中，带来影响。

I 计算物件个数（Counting Objects）

有一种物件计数问题的解法，涉及发展出一个「物件计数专用」的 class。这个 class 看起来或许很像（甚至完全像）稍早展示的 Widget class：

```
// 稍後有一些讨论，告诉你为什麼这样的设计并不很正确
class Counter {
public:
    Counter() { ++count; }
    Counter(const Counter&) { ++count; }
    ~Counter() { --count; }
    static size_t howMany()
        { return count; }
private:
    static size_t count;
};
// 以下这行仍然应该放在一个实作档中。
size_t Counter::count = 0;
```

这里的想法是，任何 classes 如果需要记录当下存在的物件个数，只需使用 Counter 来担任簿记工作即可。有两个明显的方法可以完成这项任务，其中之一是定义一个 Counter object，使它成为一个 class data member，像这样：

```
// 在需要计数的 class 中内嵌一个 Counter object。
class Widget {
public:
```

```
    .....  // Widget 该有的所有 public 成员，
           // 都放在这里。
    static size_t howMany()
    { return Counter::howMany(); }
private:
    .....  // Widget 该有的所有 private 成员，
           // 都放在这里。
    Counter c;
};
```

另一个方法是把 Counter 当做 base class，像这样：

```
// 让需要计数的 class 继承自 Counter
class Widget: public Counter {
    .....  // Widget 该有的所有 public 成员，
           // 都放在这里。
private:
    .....  // Widget 该有的所有 private 成员，
           // 都放在这里。
};
```

两种作法各有优劣。验证它们之前，我们必须注意，没有一个方法以其目前型式可以有效运作。问题在於 Counter 中的静态物件 count。这样的静态物件只有一个，但我们却需要为每一个使用 Counter 的 class 准备一个。举个例子，如果我们打算对 Widgets 和 ABCDs 计数，我们需要两个 static size_t objects，而不是一个。让 Counter::count 成为 nonstatic，并不能解决这个问题，因为我们需要的是为每个 class 准备一个计数器，而不是为每个 object 准备一个计数器。

运用 C++ 中最广为人知但名称十分诡异的一个技俩，我们就可以取得我们想要的行为：我们可以把 Counter 放进一个 template 中，然後让每一个想要使用 Counter 的 class，「以自己为 template 引数」具现出这个 template。

让我再说一次。Counter 变成一个 template：

```
template<typename T>
class Counter {
public:
    Counter() { ++count; }
    Counter(const Counter&) { ++count; }
    ~Counter() { --count; }
    static size_t howMany()
    { return count; }
private:
    static size_t count;
};
template<typename T>
size_t
Counter<T>::count = 0; // 现在这一行可以放进表头档中了。
```

於是前述的第一种实作法改变为这样：

```
// 在需要计数的 class T 中内嵌一个 Counter<T> object。
class Widget {
public:
    .....
    static size_t howMany()
    {return Counter<Widget>::howMany();}
private:
    .....
```

```
    Counter<Widget> c;
};
```

第二种作法（继承法）则改变为这样：

```
// 让需要计数的 class T 继承自 Counter<T>
class Widget: public Counter<Widget> {
    .....
};
```

注意在这两个情况中，我们是如何地将 Counter 取代为 Counter<Widget>。一如我稍早所说，每一个使用 Counter 的 class，都以它自己为引数，具现出那个 template。

「在一个 class 中，以自己为 template 引数，具现出一个 template，给自己使用」，这种策略最早系由 Jim Coplien 公开。他以多种语言（不只 C++）展示这种策略，并称此为一个「curiously recurring（诡异而循环的）template pattern」[注 1]。我不认为 Jim 故意这麼命名，不过，他对此一 pattern（样式、模式） 的描述的确比他所取的名称好得多。那可真糟糕，因为 pattern 的名称很重要，而你现在所看到的这一个名称，无法涵盖「做了什麼，如何做出」等讯息。

patterns 的命名是一种艺术，我对此并不擅长。不过我或许会把这个 pattern 称为诸如「Do It For Me」这类名称。根本上，每一个「被 Counter 产生出来的 class」，都能够针对「将 Counter 具现化」的那个 class，提供一种服务，计算现有多少个 objects。所以，class Counter<Widget> 可以计算 Widget 的物件数量，class Counter<ABCD> 可以计算 ABCD 的物件数量。

现在，Counter 成了一个 template，不论内嵌式设计或继承式设计，都可以运作，所以我们接下来面临的是，评估其相对的强度和弱点。我们的一个设计标准是，对使用者而言，「物件计数」机能应该很容易获得。上述程式码很清楚地告诉我们，继承式设计比内嵌式设计容易，因为前者只要求提及「Counter 是一个 base class」，後者却要求必须定义一个 Counter data member，并要求使用者实作出一个 howMany( ) 以唤起 Counter 的 howMany( ) [注 2]。虽然这样的工作也不是太多（客端的 howMany( ) 只需是个简单的 inline 函式），但只做一件事终究比做两件事容易些。所以让我们先把注意力放在继承式设计上。

l 使用 Public Inheritance（公开继承）

上述的继承式设计之所以能够运作，是因为 C++ 保证，每当一个 derived class object 被建构（或解构）时，其中的 base class 成份会先被建构（或後被解构）。让 Counter 成为一个 base class，便能确保：针对继承自 Counter 的 class，每当有一个 object 被产生或被摧毁，一定会有一个 Counter constructor 或 destructor 被唤起。

然而，任何时候只要牵涉 base classes 这个主题，就不要忘记 virtual destructors。Counter 应该有这样一个东西吗？良好的 C++ 物件导向设计规范说，是的，它应该有一个 virtual destructor。如果它没有，那麼当我们透过一个 base class pointer 来删除一个 derived class object 时，会导致不可预期（未有定义）的结果，而且通常是不受欢迎的：

```
class Widget: public Counter<Widget>
{ ... };
Counter<Widget> *pw =
    new Widget;  // 获得一个 base class pointer，
                 // 指向一个 derived class object。
......
delete pw; // 此将导致未知的结果 — 如果 base class
           // 缺乏一个 virtual destructor。
```

如此的行为将违反我们的需求条件：由於上述程式码没有任何不合理之处，我们的「物件计数」设计理应有绝对安全的表现。因此，这是 Counter 应该有个 virtual destructor 的强烈理由。

另一个需求条件是最佳效率（也就是不因「物件计数」而被课徵任何非必要的速度税和空间税），但我们在这里遇到一点麻烦。因为，virtual destructor（或任何虚拟函式）的出现，意味每一个 Counter（或其衍生类别）的 objects 都必须内含一个（隐藏的）虚拟

指标，而这会增加物件的大小 — 如果它们原本并没有虚拟函式的话 [注 3]。也就是说，如果 Widget 本身并无任何虚拟函式，型别为 Widget 的物件将会因为继承了 Counter<Widget> 而使大小扩张。我们不希望看到这种情况。

唯一的避免之道就是，找出一种方法，阻止客端「透过一个 base class pointer 删除一个 derived class object」。将 Counter 中的 operator delete 宣告为 private，似乎是一个合情合理的办法：

```
template<typename T>
class Counter {
public:
    .....
private:
    void operator delete(void*);
    .....
};
```

但如此一来，delete 运算式无法编译成功：

```
class Widget: public Counter<Widget> { ... };
Counter<Widget> *pw = new Widget;  ......
delete pw; // 错误。因为我们无法唤起 private operator delete
```

真是不幸。不过，真正有趣的是，new 运算式也不应该通过编译：

```
Counter<Widget> *pw =
    new Widget;  // 这一行应该无法通过编译，
                 // 因为 operator delete 是 private
```

请回忆一下稍早我对於 new, delete, exceptions（异常）的讨论，我说 C++ 的执行期系统（runtime system）有责任释放被 operator new 配置的记忆体 — 如果後继被呼叫的 constructor 失败的话。同时也请回忆一下，operator delete 是用来执行记忆体释放动作的函式。由於我们将 Counter 的 operator delete 宣告为 private，这会使得「藉由 new，将 objects 产生於 heap」的企图永远失败。

是的，这是违反直觉的，如果你的编译器不支援它，请不要惊讶。但是请注意，我所描述的行为是正确的。除此之外再无其他明显方法能够阻止「透过 Counter* pointer 删除 derived class objects」。由於我们已经拒绝「在 Counter 中设置一个 virtual destructor」的想法（它会引起非必要的空间税），所以我说，让我们放弃这个设计吧，让我们把注意力放在「使用一个 Counter data member」上面。

l 使用一个 Data Member

我们已经看过了「设置一个 Counter data member」这种设计所带来的缺点：客端必须同时定义一个 Counter data member 并撰写一个 inline 版的 howMany( )，用来呼叫 Counter 的 howMany( ) 函式。这些工作比我们希望加诸於客端程式身上的，多了一些，但它还不至於难以控制或管理。但是，除此之外还有另一个缺点：为某个 class 增加一个 Counter data member，往往会扩张其 objects 的大小。

这几乎谈不上是什麼重大的启示。毕竟，增加一个 data member 而导致 objects 的大小增加，会令你惊讶吗？但是再看一眼，再想一想，请注意 Counter 的定义：

```
template<typename T>
class Counter {
public:
    Counter();
    Counter(const Counter&);
    ~Counter();
    static size_t howMany();
private:
    static size_t count;
};
```

请注意它并没有 nonstatic data members，意味每一个型别为 Counter 的 object 其实没有内含任何东西。也许我们会以为每一个型别为 Counter 的 object，大小为 0？也许吧，但那并不正确。在这一点上，C++ 的表现相当清楚。所有 objects 都有至少 1 byte 的大小，甚至即使这些 objects 没有任何 nonstatic data members。根据这样的定义，对於具现自 Counter template 的每一个 class，sizeof 会获得某个正值。所以每一个「内含一个 Counter object」的 class 将比「不内含 Counter object」者拥有更多资料。

（有趣的是，这并不意味一个「不含 Counter」的 class，其大小就一定比「内含一个 Counter」的兄弟有更大的体积。那是因为边界排列限制（alignment restrictions）可能会造成影响。举个例子，如果 class Widget 内含两个 bytes 的资料，但系统要求必须以 4-byte 来进行边界排列，所以每一个 Widget object 将内含两个 bytes 的补白，而 sizeof(Widget) 的结果为 4。如果，就像普遍的情况那样，编译器都满足「任何物件的大小不可能为 0」这一条件，於是将一个 char 安插到 Counter<Widget> 内，那麼 sizeof(Widget) 还是传回 4，纵使 Widget 内含一个 Counter<Widget> object 亦然。那个被含入的 Counter<Widget> object 仅仅取代了原本被补白的两个 bytes 中的一个。不过，这并不是常见情节，因此我们当然不能够在设计一个「物件计数」套件时，把它放进计划内。）

我在耶诞假期开始的时候，动笔写这篇文章（正确日期是感恩节当天，这也许能够让你了解，我是如何地庆祝这个重要的节日...），现在我的情绪已经很不好了。我要做的只不过是物件计数，我再也不想要东拉西扯什麼奇怪而额外的讨论了。

Ⅰ 使用 Private Inheritance（私有继承）

再一次看看继承式设计的代码，那导致我们必须为 Counter 考虑一个 virtual destructor：

```
class Widget: public Counter<Widget>
{ ... };
Counter<Widget> *pw = new Widget;
......
delete
pw;  // 导致未定义的（未知的）结果 —
     // 如果 Counter 缺乏一个 virtual destructor。
```

稍早我们曾经试着藉由「阻止 delete 运算式顺利编译」而阻止这一系列动作，但是我们发现，那同时也阻止了 new 运算式的顺利编译。除此之外，还有其他某些东西也是我们可以禁止的。我们可以禁止一个 Widget* pointer（这是 new 的回传值）被隐式转型为一个 Counter<Widget>* pointer。换句话说，我们可以阻止继承体系中的指标型别转换。我们唯一需要做的就是将「public 继承」改为「private 继承」：

```
class Widget: private Counter<Widget>
{ ... };
Counter<Widget> *pw =
    new Widget;  // 错误! 没有隐式转换函式（implicit conversion）可以
                 // 将 Widget* 转为 Counter<Widget>*
```

此外，我们很开心地发现，以 Counter 做为 base class，并不会增加 Widget 的大小 — 如果和 Widget 独立个体的大小相比的话。是的，我知道我才刚刚告诉过你，没有任何 class 的大小为 0，但是 — 唔，那并不是我真正的意思。 我的真正意思是，没有任何一个 objects 的大小为 0。C++ 标准规格说得很清楚，一个 derived object 之中的「base-class 成份」的大小可以是 0。事实上，许多编译器都发展出所谓的「空白基础类别最佳化技术」（empty base optimization） [注 4]。

因此，如果一个 Widget 内含一个 Counter，Widget 的大小一定会增加。因为 Counter 的 data member 完全属於自己，而不是别人的 base-class 成份，因此它必须有非零大小。但如果 Widget 继承自 Counter，编译器便得以将 Widget 的大小保持在原先状态。这个事实为那些「记忆体使用状态非常紧绷而类别设计中涉及空白基础类别」的设计，提出了一个有趣的规则：当「private 继承」和「复合技术（containment, composition）」都能完成

相同目的时，尽量选用「private 继承」。（译注：这一点乍见之下和 Scott Meyers 的《Effective C++ 2/e》条款 42 有所抵触。该条款最後一段建议大家，如果「private 继承」和「复合技术」都能完成相同目的，尽量选用复合技术。然而请你注意，本文所给的这个建议是有前提的。）

最後的设计几近完美。它实践了效率的要求，前提是你的编译器具有「空白基础类别最佳化」（empty base optimization）的能力，如此一来「继承自 Counter」这一事实才不会增加下层类别的物件大小。此外，所有的 Counter member functions 都必须是 inlin 函式。这样的设计也实践了安全需求，因为计数动作是由 Counter member functions 自动处理，那些函式会自动被 C++ 呼叫，而 private 继承机制的使用则阻止了隐式转型 — 「隐式转型」允许 derived-class objects 被当做 base-class objects 一样地处理。（好吧，我承认，它并非绝对安全：Widget 的作者可能荒谬地以一个 Widget 以外的类别来具现化 Counter，也就是说，他可能让 Widget 继承自 Counter<Gidget>。我对这种可能性所采取的态度是：不加理会。）

这样的设计对客端而言很容易使用，但是可能有人会咕哝说，还可以更简单。使用 private 继承机制，意味 howMany( ) 会在衍生类别中成为 private，所以衍生类别中必须含入一个 using declaration，使 howMany( ) 成为 public，才能被客端所用：

```
class Widget: private Counter<Widget> {
public:
    // 让 howMany 成为 public
    using Counter<Widget>::howMany;
    ..... // Widget 的剩馀部份没有改变。
};
class ABCD: private Counter<ABCD> {
public:
    // 让 howMany 成为 public
    using Counter<ABCD>::howMany;
    ..... // ABCD 的剩馀部份没有改变。
};
```

对那些并不支援 namespaces（命名空间）的编译器而言，以上目的也可以改用旧有的存取层级来完成（但并不被鼓励）：

```
class Widget: private Counter<Widget> {
public:
    // 让 howMany 成为 public
    Counter<Widget>::howMany;
    ..... // Widget 的剩馀部份没有改变。
};
```

至此，有必要执行「物件计数」的那些客端程式，以及有必要让该计数器为其客户所用（亦即成为 class 介面的一份子）的 classes，必须做两件事情：将 Counter 宣告为一个 base class 并让 howMany( ) 可被取用 [注 5]。

然而，继承机制的使用，会导致两个值得注意的情况。第一件事是模棱两可（ambiguity）。假设我们打算对 Widgets 计数，而我们希望让这个计数值供一般运用。一如先前所展示，我们令 Widget 继承自 Counter<Widget>，并令 Widget::howMany( ) 成为 public。 现在假设我们有一个 class SpecialWidget，以 public 方式继承自 Widget，我们希望提供给 SpecialWidget 使用者一如 Widget 使用者所能享受的机能。没问题，只需令 SpecialWidget 继承自 Counter<SpecialWidget> 即可。

但这里出现了模棱两可（ambiguity）的问题。哪一个 howMany( ) 对 SpecialWidget 而言才是可用的呢？是继承自 Widget 的那个，或是继承自 Counter<SpecialWidget> 的那个？我们所希望的，当然是来自 Counter<SpecialWidget> 的那个，但是我们没办法在未明确写出 SpecialWidget::howMany( ) 的情况下说出我们的心愿。幸运的是，它只是一个简单的 inline 函式：

```
class SpecialWidget: public Widget,
```

```
                private Counter<SpecialWidget> {
public:
    .....
    static size_t howMany()
    { return Counter<SpecialWidget>::howMany(); }
    .....
};
```

关於「使用继承机制来完成物件计数工作」的第二个意见是，**Widget::howMany( )** 传回的值不只包括 **Widget objects** 的个数，也包括 **Widget** 衍生类别所产生的 **objects**。如果 **Widget** 的唯一衍生类别是 **SpecialWidget**，而一共有五个 **Widget** 独立物件和三个 **SpecialWidgets** 独立物件，那麼 **Widget::howMany( )** 将传回 **8**。毕竟，每一个 **SpecialWidget** 的建构，也同时会完成其基础类别（**Widget** 成份）的建构。

▌摘要

以下数点是你需要记住的：

- 物件计数工作的自动化并不困难，但也并非直观想像中的那麼简单。运用 **"Do It For Me" pattern**（**Coplien** 所谓的 **"curiously recurring template pattern"**）便有可能产生正确数量的计数器。运用 **private** 继承机制，可以提供物件计数能力，而又不扩张物件的大小。

- 当客端有机会选择「继承自一个 **empty class**」或「内含某个 **class object** 做为 **data member**」时，继承是比较好的选择，因为它允许更紧密的物件。

- 由於 **C++** 尽一切努力要在 **heap objects** 建构动作失败时避免发生记忆体漏洞（**memory leaks**），所以凡是需要用到 **operator new** 之程式码，通常也需要用到对应的 **operator delete**。

- **Counter class template** 并不在乎你是否继承它，或内含它的一个 **object**。它看起来都一样。因此，客端可以自由选择使用「继承机制」或「复合（组合）技术」，甚至在同一个应用程式或程式库的不同地点使用不同的策略。

▌注解与叁考资料

[1] James O. Coplien. "The Column Without a Name: A Curiously Recurring Template Pattern," C++ Report, February 1995.

[2] 另一种方法是忽略 **Widget::howMany( )**，让客端直接呼叫 **Counter<Widget>::howMany( )**。然而，对本文目的而言，我们将假设我们希望 **howMany( )** 是 **Widget** 介面的一部份。

[3] Scott Meyers. More Effective C++ (Addison-Wesley, 1996), pp. 113-122.

[4] Nathan Myers. "The Empty Member C++ Optimization," Dr. Dobb's Journal, August 1997。可自以下网站获得：
http://www.cantrip.org/emptyopt.html.

[5] 只要对这个设计做一点简单的变化，就可以让 **Widget** 以 **Counter<Widget>** 计算物件个数，并且不让这个计数值被 **Widget** 的客户所用，甚至不允许 **Counter<Widget>::howMany( )** 被直接呼叫。下面这个练习留给时间充裕的读者：继续讨论更多变化。

▌进一步的读物

如果想要学习更多关於 **new** 和 **delete** 的细节，请阅读 **Dan Saks** 在 **CUJ 1997** 年一月至七月所主持的专栏，或是我的 **More Effective C++ (Addison-Wesley, 1996)** 条款 **8**。如果想要更广泛地验证物件计数（**object-counting**）问题，包括如何限制某个 **class** 被具现化的次数，请看 **More Effective C++** 条款 **26**。

▌致谢

Mark Rodgers, Damien Watkins, Marco Dalla Gasperina, 和 Bobby Schmidt 针对本文草稿提出了一些意见。他们的洞见和提议，使本文在许多方面有了更好的改善。

作者

Scott Meyers 是畅销书籍 Effective C++ 第二版和 More Effective C++ 的作者（两本书都由 Addison Wesley 出版）。你可以从 http://www.aristeia.com中找到更多有关於他、他的书、他的那只狗的讯息。译注：《Effective C++》第二版以 Meyers 的狗为封面。

译者

陈崴，自由撰稿人，专长 C++/Java/OOP/GP/DP。惯以热情的文字表现冰冷的技术，以冷冽的文字表现深层的关怀。

sidebar : Placement new 与 placement delete

malloc( ) 在 C++ 中的对等物是 operator new，free( ) 在 C++ 中的对等物则是 operator delete。和 malloc( ) 及 free( ) 不同的是是，operator new 和 operator delete 都可以被重载，重载後的版本可接受与母版不同个数、不同型别的叁数。这对 operator new 来说一向正确，但直到最近，才对 operator delete 也成立。

operator new 的正常标记（signature）是：

```
void * operator new(size_t) throw (std::bad_alloc);
```

（从现在起，为了简化，我将刻意忽略 exception specifications（译注：就是上述的 throw (std::bad_alloc)），因为它们和我目前要说的重点没有什麽密切关系。）operator new 的重载版本只能增加新叁数，所以一个 operator new 重载版本可能长这个样子：

```
void * operator new(size_t, void *whereToPutObject)
{ return whereToPutObject; }
```

这个特殊版本的 operator new 接受一个额外的 void* 引数，指出此函式应该回传什麽指标。由於这个特殊形式在 C++ 标准程式库中是如此常见而有用（宣告於表头档 <new>），因而有了一个属於自己的名称："placement new"。这个名称表现出其目的：允许程序员指出「一个 object 应该诞生於记忆体何处」。

随着时间过去，任何「要求额外引数」的 operator new 版本，也都渐渐采用 placement new 这个术语。事实上这个术语已经被铭记於 C++ 标准规格中。因此，当 C++ 程序员谈到所谓的 placement new 函式，他们所谈的可能是上述那个「需要额外一个 void* 叁数，用以指出物件置於何处」的版本，但也可能是指那些「所需引数比单一而必要之 size_t 引数更多」的任何 operator new 版本，包括上述函式，也包括其他「引数更多」的 operator new 函式。

换句话说，当我们把焦点集中在记忆体配置时，"placement new" 意味「operator new 的某个版本，接受额外引数」。这个术语在其他场合可能有其他意义，但我们不需继续深入，所以，到此为止。如果你需要更多细节，请叁考本文最後所列的叁考读物。

和 placement new 类似，术语 "placement delete" 意味「operator delete 的某个版本，接受额外引数」。operator delete 的「正常」标记如下：

```
void operator delete(void*);
```

所以，任何版本的 operator delete，只要接受的引数多於上述的 void*，就是一个 placement delete 函式。

现在让我们重回本文所讨论的一个主题。当 heap object 在建构期间丢出一个异常，会发生什麽事？再次考虑以下这个简单例子：

```
class ABCD { ... };
ABCD *p = new ABCD;
```

假设产生 ABCD object 时导致了一个异常。前列的主文内容指出，如果异常来自 ABCD 建构式，operator delete 会自动被唤起，释放 operator new 所配置的记忆体。但如果 operator new 被多载化，情况将如何？如果不同版本的 operator new 以不同的方式配置记忆体，情况又将如何？operator delete 如何知道该怎麽做才能正确释放记忆体？此外，如果 ABCD object 系以 placement new 产生出来（像下面这样），又该如何：

```
void *objectBuffer = getPointerToStaticBuffer();
ABCD *p = new (objectBuffer) ABCD; // 在一个静态缓冲区中产生一个 ABCD object
```

上述那个 **placement new** 并不配置任何记忆体。它只是传回一个指标，指向那个它所接受的静态缓冲区。也因此，不需要任何释放动作。

很显然，**operator delete** 所采取的行动（用以回复其对应之 **operator new** 的行为）必须视配置记忆体时所采用的 **operator new** 版本而定。

为了让程式员有机会指示「如何回复某个特殊版本之 **operator new** 的行为」，**C++** 标准委员会扩展了 **C++**，允许 **operator delete** 也能够被多载化。当 **heap object** 的 **constructor** 丢出一个异常，整个游戏便改走另一条路，呼叫起特殊的 **operator delete** 版本，此一版本带有额外叁数型别，这些型别将对应於先前被唤起之 **operator new** 版本。

如果没有任何一个版本的 **placement delete** 的额外叁数能够对应於「被唤起之 **placement new** 的额外叁数」，那麼，就不会有任何 **operator delete** 被唤起。於是，**operator new** 的行为所带来的影响就无法被抹除。对於那些「**placement** 版」的 **operator new**，这没问题，因为它们并不真正配置记忆体。然而，一般而言，如果你产生一个自定的「**placement** 版」的 **operator new**，你也应该产生一个对应的自定的「**placement** 版」**operator delete**。啊呀，大部份编译器都还没有支援 **placement delete**。这种编译器所产生出来的程式码，使你几乎总得蒙受一个记忆体漏洞（**memory leak**）。如果在 **heap object** 建构期间有一个异常被丢出，因为不会有任何人企图释放 **constructor** 被唤起之前被配置的记忆体。

译注：根据我的测试，**GNU C++ 2.9, Borland C++Builder 40, Microsoft Visual C++ 6.0** 三家编译器都已经支援 **placement** 版本的 **operator delete**。其中以 **Visual C++** 最为体贴，当「没有任何一个版本的 **placement delete** 的额外叁数能够对应於被唤起之 **placement new** 的额外叁数」时，**Visual C++** 会给你一个警告讯息：

no matching operator delete found; memory will not be freed if initialization throws an exception.

9.4 为智能指标实作 operator->*（Implementing operator->* for Smart Pointers）译者：陈崴

侯捷注：本文系北京《程序员》杂志 **2001/09** 的文章。译笔顺畅，技术饱满。承译者陈崴先生与《程序员》杂志负责人蒋涛先生答允，转载於此，以飨台湾读者，非常感谢。未得陈崴先生与蒋涛先生二人之同意，任何人请勿将此文再做转载。

译注：以下是本文所采用的几个特别请读者注意的术语：

client： 客端。
type： 型别。为避免和其他近似术语混淆，本文译为「型别」而非「类型」。
instantiated: 具现化。「针对一个 **template**，具体实现出一份实体」的意思。
instance： 实体。「案例」是错误的译法。
parameter： 叁数。或称型式叁数，形叁。
argument： 引数。或称实质叁数，实叁。
user-defined: 使用者自定

至於 **class, object, data member, member function, constructor, destructor, template** 等术语，皆保留不译。

当我撰写 **More Effective C++: 35 Ways to Improve Your Programs and Designs (Addison-Wesley, 1995)** 时，我在其中探讨了一个主题：智能指标（**smart pointers**）。结果，我收到许多询问，其中最有趣的一个问题来自 **Andrei Alexandrescu**，他问到：『一个真正很有智能的智能指标，该不该对 **operator->*** 实施多载化呢？我从未看过有人那麼做』。是啊，我也从未看过有人那麼做，所以我决定试试。我所获得的结果深具启发意义，远比「实现 **operator->***」更多，其中涉及极有趣也极有用的 **templates** 运用方式。

I 检阅 operator->*

如果你和绝大多数程式员一样，你大概不曾用过 **operator->***。因此，在我解释如何针对智能指标实作这个 **operator** 之前，我得先带你检阅一下它的内建行为。

假设有一个 **class C**，一个指标 **pmf** ，用以指向 **C** 的无叁数 **member function**，以及一个指标 **pc**，用以指向 **C object**。以下运算式：

```
(pc->*pmf)();   // 在 *pc 身上唤起 member function *pmf。
```

会在「**pc** 所指的那个 **object**」身上唤起 **pmf** 所指的那个 **member function**。一如程式列表一所示，一个指向 **member functions** 的指标，其行为类似指向一般函式的指标，只是语法稍稍复杂些而已。透过这种方式来写码，环绕 **pc->*pmf** 的小括号是必要的，因为编译器会把以下式子：

```
pc->*pmf();   // 错误!
```

视为：

```
pc->*(pmf()); // 错误!
```

程式列表一

```
class Wombat {          // wombats 是澳洲特产的一种有袋动物，
public:                 // 看起来有点像狗。
    int dig();          // 传回 depth dug
    int sleep();        // 传回 time slept
};
typedef int (Wombat::*PWMF)(); // PWMF：一个指向 Wombat member function 的指标。
Wombat *pw = new Wombat;
PWMF pmf = &Wombat::dig;   // 令 pmf 指向 Wombat::dig
(pw->*pmf)();              // 这和 pw->dig(); 相同。
pmf = &Wombat::sleep;      // 令 pmf 指向 Wombat::sleep
(pw->*pmf)();              // 这和 pw->sleep(); 相同。
```

为支援 operator->* 而做的设计

就像许多运算子一样，**operator->*** 也是二元的：它接受两个引数。当我们针对智能指标实作 **operator->***时，其左引数是个智能指标，指向一个型别为 **T** 的物件。其右引数是个指标，指向 **class T** 的一个 **member function**。呼叫 **operator->***之後，我们对其回传值唯一能做的事情就是给它一个叁数列，使它成为一个「函式呼叫」。所以，**operator->*** 的回传型别必须是某种「**operator()**（所谓 **function call operator**）得以施行於上」的东西。**operator->*** 的回传值代表一个审理中的、未完成的函式呼叫，所以我把 **operator->*** 传回的物件型别称为 **PMFC**，意思是一个 **"Pending Member Function Call."**

把这些整合在一起，於是我们获得程式列表二的虚拟码（**pseudocode**）：

程式列表二（pseudocode）

```
class PMFC {                // "Pending Member Function Call"
public:
    ...
    return type operator()( parameters ) const;     // 译注(1)
    ...
};
template<typename T>        // template，用於 T 智能指标，可以支援 operator->*。
class SP {
public:
    ...
    const PMFC operator->*( return type (T::*pmf)( parameters ) ) const;
        // 译注(2)
    ...
};
// 译注：上述(1)的 return type 和 parameters 必须和 (2) 中的对应物完全一样。
```

由於每一个 PMFC object 都代表着一个未完成的呼叫：呼叫 operator->* 所接收到的 member function，因此 member function 和 PMFC::operator() 所期待的叁数列完全相同。为了简化事情，我假设 T 的 member functions 不接受任何引数（稍後我会移除这项限制），这意味你可以将程式列表二重新定义为程式列表三。

■ 程式列表三

```
class PMFC {
public:
    ...
    return type operator()() const;
    ...
};
template<typename T>
class SP {
public:
   ...
    const PMFC operator->*( return type (T::*pmf)() ) const;
    ...
};
```

上述「pmf 所指向之 member function」的回传型别究竟会是什麼呢？可能是 int, double, 也可能是 const Wombat&。是的，它可能是任何型别。你应该将这种无穷可能的组合以 template 来表现。於是，operator->* 现在变成了一个 member function template。此外，PMFC 也变成了一个 template，因为不同的「operator->* 具现体」会传回不同型别的 PMFC objects。每个 PMFC object 在其 operator() 被唤起时，都必须知道回传型别是什麼。

一旦将它模板化（templatization）之後，你便可以舍弃虚拟码（pseudocode），写出真正的 PMFC 和 SP::operator->*，如程式列表四。

■ 程式列表四

```
template<typename ReturnType>   // template，用於一个「回传型别为 ReturnType」的
class PMFC {                    // 未完成的 member function call。
public:
    ...
    ReturnType operator()() const;
    ...
};
template<typename T>
class SP {
public:
    ...
    template<typename ReturnType>
        const PMFC<ReturnType>
            operator->*( ReturnType (T::*pmf)() ) const;
    ...
};
```

■ 无任何叁数的 Member Functions

身为一个未完成的 member function call，PMFC 需要知道两样东西，才能实作其 operator()：(1) 它究竟要呼叫哪个 member function，(2) 在哪个 object 身上唤起该 member function。PMFC constructor 应该是索取这些引数的一个合适地点。此外，将上述两样东西放在一个标准的 pair object 内似乎也是个好主意。於是我们获得程式列表五。

■ 程式列表五

```
template<typename ObjectType,        // 提供 mem func 的那个 class。
```

```
        typename ReturnType,        // mem func 的回传型别。
        typename MemFuncPtrType>    // mem func 的完整标记。
class PMFC {
public:
    typedef std::pair<ObjectType*, MemFuncPtrType> CallInfo;
    PMFC(const CallInfo& info): callInfo(info) {}
    ReturnType operator()() const
        { return (callInfo.first->*callInfo.second)(); }
private:
    CallInfo callInfo;
};
```

乍见之下虽然似乎很复杂，其实相当简单。当你产生一个 **PMFC**，你必须指出要呼叫哪一种 **member function**（译注：这就是上述 **MemFuncPtrType** 的作用），并指出在哪一种 **object** 身上唤起它（译注：这就是上述 **ObjectType** 的作用）。稍後当你唤起 **PMFC** 的 **operator()** 函式，後者所唤起的便是这个「型别已被完整储存（记录）下来」的 **member function**。

请注意上述的 **operator()** 系根据语言内建的 **operator->*** 实作出来。要知道，只有当智能指标的「使用者自定之 **operator->***」被呼叫，才会产生出一个 **PMFC objects**，这暗喻「使用者自定之 **operator->***」应该根据「语言内建的 **operator->***」实作出来。这和我们所熟知的以下事实是一个很好的对称：「使用者自定之 **operator->**」的行为，和「语言内建之 **operator->**」彼此关连，因为 **C++** 对於「使用者自定之 **operator->**」的每一个呼叫，最终都是（隐喻地）呼叫到「语言内建的 **operator->**」。如此的对称现象令人安心，表示目前的设计走在正确的方向上。

你可能已经注意到了，**ObjectType**，**ReturnType**，和 **MemFuncPtrType** 三个 **template parameters** 似乎有点赘馀。因为如果我们拥有 **MemFuncPtrType**，我们就能够推演出 **ObjectType** 和 **ReturnType**。毕竟不论 **ObjectType** 或 **ReturnType**，都是 **MemFuncPtrType** 的一部份。的确，我们能够使用「模板偏特化」（**partial template specialization**）技术，从 **MemFuncPtrType** 推导出 **ObjectType** 和 **ReturnType**，但由於支援「偏特化」技术之商用编译器还不十分普及（译注：例如 **Visual C++ 6** 就没有支援），所以我没有选择这种作法。如果想基於「偏特化技术」来进行设计，相关细节请看稍後的「**Partial Template Specialization** 与 **operator->***」方块文字。

有了程式列表五中的 **PMFC** 实作内容，**SP<T>** 的 **operator->*** 几乎已经完成。它所传回的 **PMFC object** 需要一个 **object pointer** 和一个 **member function pointer** 做为初值。一如程式列表六所示，智能指标习惯上会有一个 **data member** 用来存放 **object pointer**；至於 **member function pointer**，其实就是 **operator->*** 所接获的引数。因此，程式列表七的代码应该可以有效运作。以我所测试的两个编译器（**Visual C++ 6** 和 **egcs 1.1.2**）而言，的确如此。

I 程式列表六

```
template <typename T>
class SP {
public:
    SP(T *p): ptr(p) {}
    template <typename ReturnType>
        const PMFC<T, ReturnType, ReturnType (T::*)()>
            operator->*(ReturnType (T::*pmf)()) const
                { return std::make_pair(ptr, pmf); }
    ...
private:
    T* ptr;
};
```

I 程式列表七

```
#include <iostream>
```

```
#include <utility>
using namespace std;
template<typename ObjectType, typename ReturnType, typename MemFuncPtrType>
class PMFC { ... };              // 一如以往
template <typename T>            // 一如以往
class SP { ... };
class Wombat {
public:
    int dig()
    {
        cout << "Digging..." << endl;
        return 1;
    }
    int sleep()
    {
        cout << "Sleeping..." << endl;
        return 5;
    }
};
int main()
{                                    // 一如以往，PWMF 是个指标，
    typedef int (Wombat::*PWMF)();   // 指向一个 Wombat member function
    SP<Wombat> pw = new Wombat;
    PWMF pmf = &Wombat::dig;   // 令 pmf 指向 Wombat::dig
    (pw->*pmf)();              // 唤起我们的 operator->*;
                               // 印出 "Digging..."
    pmf = &Wombat::sleep;      // 令 pmf 指向 Wombat::sleep
    (pw->*pmf)();              // 唤起我们的 operator->*;
}                              // 印出 "Sleeping..."
```

是的，我发现了，这份程式码会出现资源漏洞（resource leak），因为被 new 出来的 Wombat 从未被 delete。此外，这份程式码使用了一个 using directive（也就是 using namespace std;），而其实使用 using declarations 也就够了。请试着把焦点放在 SP::operator->* 和 PMFC 的互动上，不要太拘泥如此小节。如果你了解 (pw->*pmf)() 的行为由来，毫无问题你的实力必可轻易修正上例关於编程风格上的缺失。

顺带一提，由於 operator->* member functions 和所有的 PMFC member functions 都是 inline 函式（虽然并无明显宣告），因此，你可以预期，针对以下述句所产生出来的码：

```
(pw->*pmf)();
```

其中运用 SP 和 PMFC 的情况，将和以下等值述句所产生出来的码没有两样：

```
(pw.ptr->*pmf)();
```

其中只使用内建运算动作。因此，使用 SP's 的多载化 operator->* 和 PMFC's 的多载化 operator()，执行期成本可以为零：程式码的额外成本为 零，资料的额外成本也为零。当然，实际成本视你的编译器的最佳化能力、你的标准程式库对於 pair 和 make_pair 的实作而定。以我所测试的两个编译器（及其附带的标准程式库）而言，打开所有最佳化选项後，有一个编译器会获得执行期零成本的 operator->* 实作码，另一个不会。

l 加上对 const Member Functions 的支援

请仔细看看 SP<T> 的 operator->* 函式的正式叁数：ReturnType (T::*pmf)()。让我更明确地提示你，它不是 ReturnType (T::*pmf)() const。这意味，没有任何一个「指向 const member function」的指标可以被传递给 operator->*，这也就意味 operator->* 不支援 const member functions。如此露骨的不公平待遇在一个具有良好设计的软体系统中

是毫无地位的。幸运的是，我们很容易破除这项限制。只要为 SP 加上第二个 operator->* template，整份设计就可适用於「指向 const member functions」的指标；见程式列表八。有趣的是，PMFC 的内容不需有任何改变，因为其型别参数 MemFuncPtrType 会系结到任何一个 member function pointer 型别身上，不论那个 member function 是否为 const。

程式列表八

```
template <typename T>
class SP {
public:
    ...                     // 如前
    template <typename ReturnType>
        const PMFC<T, ReturnType, ReturnType (T::*)() const>  // 加上 const
            operator->*(ReturnType (T::*pmf)() const) const   // 加上 const
                { return std::make_pair(ptr, pmf); }
    ...                     // 如前
};
```

支援「拥有参数的 Member Functions」

有了零参数的经验在手，我们接下来设法支援「一个指标，指向 member function，该 member function 拥有一个参数」。整个过程出乎意料地简单，因为你唯一需要做的就是修改 operator->* 所接受的参数型别，然後透过 PMFC 传播这项改变。事实上，你唯一需要做的就是为 operator->* 增加一个新的 template 参数，其型别应该是被指向的 member function 所能（所愿）接受的参数型别。然後修改其他每一样需要与此保持一致性的东西。此外，由於 SP<T> 也应该同时支援「接受零个参数」的 member functions 和「接受一个参数」的 member functions，所以你应该为它添加一个新的 operator->* template。在程式列表九中，我只展示对於 nonconst member functions 的支援，事实上对 const member functions 的支援也应该同时完成。

程式列表九

```
template <typename ObjectType, typename ReturnType, typename MemFuncPtrType>
class PMFC {
public:
    typedef pair<ObjectType*, MemFuncPtrType> CallInfo;
    PMFC(const CallInfo& info)
   : callInfo(info) {}
    // 支援 0 个参数
    ReturnType operator()() const
        { return (callInfo.first->*callInfo.second)(); }
    // 支援 1 个参数
    template<typename Param1Type>
    ReturnType operator()(Param1Type p1) const
            { return (callInfo.first->*callInfo.second)(p1); }
private:
    CallInfo callInfo;
};
template <typename T>
class SP {
public:
    SP(T *p): ptr(p) {}
    // 支援 0 个参数
    template <typename ReturnType>
        const PMFC<T, ReturnType, ReturnType (T::*)()>
            operator->*(ReturnType (T::*pmf)()) const
```

```
            { return std::make_pair(ptr, pmf); }
    // 支援 1 个参数
    template <typename ReturnType, typename Param1Type>
        const PMFC<T, ReturnType, ReturnType (T::*)(Param1Type)>
            operator->*(ReturnType (T::*pmf)(Param1Type)) const
                { return std::make_pair(ptr, pmf); }
    ...
private:
    T* ptr;
};
```

一旦你能够处理零个参数和一个参数，就很容易处理任意个数的参数。让我做个整理，为了支援「具有 n 个参数」的 member functions，请在 SP 中宣告两个 member template operator->*，一个用来支援 nonconst member functions，一个用来支援 const 版本。每一个 operator->* template 都应该获得 n+1 个型别参数，其中 n 个做为函式参数使用，剩馀那个用来指定函式回传型别。接下来，在 PMFC 中加上对应的 operator() template，就大功告成了。

l 将所有对 operator->* 的支援工作包装起来

许多应用软体对智能指标有多种不同的变化运用，如果他们必须针对每一种应用，重复上述工作，他们一定不会高兴。如果你想看看智能指标的变化应用，包括像杀手一般酷的 C++ 程式，请看 Kevin S. Van Horn 的网站 http:// www.xmission.com/ ~ksvsoft/code/smart_ptrs.html。幸运的是，对 operator->* 的所有支援工作，可以被包装为一个 base class 型式，如程式列表十。

l 程式列表十

```
template <typename T>   // 这是一个 base class，用来包装智能指标
class SmartPtrBase {    // 对 operator->* 的支援。
public:
    SmartPtrBase(T *initVal): ptr(initVal) {}
    // 支援 0 个参数。
    template <typename ReturnType>
        const PMFC<T, ReturnType, ReturnType (T::*)()>
            operator->*(ReturnType (T::*pmf)()) const
                { return std::make_pair(ptr, pmf); }
    // 支援 1 个参数。
    template <   typename ReturnType, typename Param1Type>
        const PMFC<T, ReturnType, ReturnType (T::*)(Param1Type)>
            operator->*(ReturnType (T::*pmf)(Param1Type)) const
                { return make_pair(ptr, pmf); }
    ...
protected:
    T* ptr;
};
```

凡是希望提供 operator->*支援能力的智能指标，只需继承 SmartPtrBase 即可。但是，只有当「智能指标内含一个一般指标，用以完成真正的指向动作」时，上述设计才适用。「内含一般指标」是最常见的一种智能指标设计方式，但还有其他不同的设计，这类不同的设计可能需要以不同於此处所描述的方式，将 operator->* 机能包装起来。其中最好的方式或许是利用 private 继承机制，因为如果使用 public 继承机制，便有必要为 SmartPtrBase 加上一个 virtual destructor，因而增加其大小（以及其所有 derived classes 的大小）。Private 继承机制可以避免付出这样的代价，虽然，它需要一个 using declaration（见程式列表十一）以便让「以 private 方式继承而来的 operator->* templates」成为 public。

如果要把它们包装得更好，可以将 SmartPtrBase 和 PMFC template 都放进一个命名空间（namespace）中。

l 程式列表十一

```
template <typename T>
class SP: private SmartPtrBase<T> {
public:
    SP(T *p ): SmartPtrBase<T>(p) {}
    using SmartPtrBase<T>::operator->*;     // 让「private 继承而来的
                                            // operator->* templates」成为 public。
    // 智能指标的一般函式放在这儿。operator->* 机能则是藉由继承机制获得。
};
```

l 松散的结尾

在我完成针对智能指标而做的 operator->* 支援工作後，我把我的解法公布於 Usenet 的消息群组 comp.lang.c++.moderated，看看我是否遗漏了什麼。很快地 Esa Pulkkinen 做出了以下言论：

你的方法至少有两个问题：

1. 你无法使用 pointers to data members（虽然这很容易解决）。
2. 你无法使用「使用者自定（user-defined）」的 pointers-to-members。如果有人多载化了 operator->*，令它接受「行为类似 member pointers」的某些 objects，你可能会想要在你的 smart pointer class 中支援如此的 "smart pointers to members" 。不幸的是，你需要 traits classes 才能获得这种多载化後的 operator->* 的回传型别。

Smart pointers to members! 喔欧，Esa 是对的。（事实上比我最初所了解的还要正确。就在撰写本文草稿之後的很短时间内，我的一个客户对我展现了一个问题，该问题可以使用 smart pointers to members 自然地解决掉。我也很惊讶。）幸运的是，本文已经够长了，我可以在这里停下来，把解决 Esa 的评论的工作留给读者。我决定这麼做。

l 摘要

如果你的目标是要让你的智能指标（smart pointers）的行为尽可能相容於内建（原生）指标，你应该支援 operator->*，就像内建指标的作为一样。运用 class templates 和 member templates，我们得以比较轻松地实作出如此的支援能力。把这些实作码包装为一个 base class，可以增加其复用性，方便其他智能指标的设计者。

l 致谢

Andrei Alexandrescu 除了引起我对 operator->* 的最初兴趣，也帮我简化了我对 PMFC 的实作。Andrei 还针对本文早期草稿提供了极富深刻意义的见解，并附带原始码。Esa Pulkkinen 和 Mark Rodgers 也做了相同的贡献。他们对本文提供的协助，使我受惠良多。

l 作者

Scott 是一位 C++ 顾问，也是 Effective C++ CD-ROM, Effective C++, 和 More Effective C++ 的作者。你可以透过 http://www.aristeia.com/ 和他联络。

l 译者

陈崴，自由撰稿人，专长 C++/Java/OOP/GP/DP。

l sidebar : Partial Template Specialization 与 operator->*

在我撰写本文期间，Esa Pulkkinen 和 Mark Rodgers 告诉我，模板偏特化技术（partial template specialization）可用来从一个「 pointer to member function」型别中萃取出该 member function 所属的 class 型别，以及函式回传型别。唯一需要的是施行 traits 技术，那是一种被广泛运用於 C++ 标准程式库的技术。关於 traits 技术，请叁考 Nathan Myers 的文章："Traits: A New and Useful Template Technique," C++ Report, June 1995，此文也出现於 http://www.cantrip .org/traits.html。

Mark Rodgers 建议程式列表十二所列的那些 templates，它们用来接受零个叁数或一个叁数的 member functions。要将它们扩充为接受更多叁数，当然毫无问题而且作法十分直接了当。有了这些 templates，PMFC 就可以简化为「只需一个型别叁数，亦即

MemFuncPtrType」。那是因为，另两个型别参数 ObjectType 和 ReturnTypecan 可根据 MemFuncPtrType 推导出来：

- ObjectType 是 MemFuncTraits<Mem FuncPtrType>::ObjectType.
- ReturnType 是 MemFuncTraits<Mem FuncPtrType>::ReturnType.

■ 程式列表十二

```
template <typename T>                          // traits class
struct MemFuncTraits {};
template <typename R, typename O>              // partial specialization
struct MemFuncTraits<R (O::*)()> {             // for zero-parameter
    typedef R ReturnType;                      // non-const member
    typedef O ObjectType;                      // functions
};
template <typename R, typename O>              // partial specialization
struct MemFuncTraits<R (O::*)() const> {       // for zero-parameter
    typedef R ReturnType;                      // const member
    typedef O ObjectType;                      // functions
};
template <typename R, typename O, typename P1>  // partial specialization
struct MemFuncTraits<R (O::*)(P1)> {            // for one-parameter
    typedef R ReturnType;                       // non-const member
    typedef O ObjectType;                       // functions
};
template <typename R, typename O, typename P1> // partial specialization
struct MemFuncTraits<R (O::*)(P1) const> {     // for one-parameter
    typedef R ReturnType;                      // const member
    typedef O ObjectType;                      // functions
};
```

运用这些 templates，我们便可获得程式列表十三中新的 PMFC。我不打算在这里告诉你 traits 的技术细节，也不打算解释为什麼 typename 必须出现在 template 内的型别名称之前。上述这些 templates 可以大量降低 smart pointer classes「欲支援 operator->*」时的工作量。事实上，Mark Rodgers 提醒了我们，单单一个 operator->* template 就可以支援所有可能的 member function pointers，不论後者所指向的 member functions 接受多少个参数，也不论那些 member functions 是否为 const。是的，只要将 SP（或 SmartPtrBase）中所有的 operator->* templates 全部取代为程式列表十四的码即可。型别参数 MemFuncPtrType 将被系结於任何一种 pointer to member function 型别身上，不论该 member function 的参数个数、回传型别、常数性（constness）。整个 pointer to member function 型别会被传递给 PMFC，在那里，偏特化机制（partial specialization）会起而行，将我们所在乎的各个型别抓出来。

■ 程式列表十三

```
template <typename MemFuncPtrType>
class PMFC {
public:
    typedef typename MemFuncTraits<MemFuncPtrType>::ObjectType ObjectType;
    typedef typename MemFuncTraits<MemFuncPtrType>::ReturnType ReturnType;
    ...                 // 一如以往
};
```

■ 程式列表十四

```
template <typename MemFuncPtrType>
const PMFC<MemFuncPtrType>
```

```
operator->*(MemFuncPtrType pmf) const
{ return std::make_pair(ptr, pmf); }
```
陈崴注 1：完整程式码如下，并附个人测试心得：
```
// vc6[x]  cb4[x]  gcc2.91.57_on_win32[o]
#include <iostream>
#include <utility>   // for pair and make_pair()
using namespace std;
// partial specialization 技术只有 GCC 和 BCB 支援。VC6 失败。
template <typename T>
struct MemFuncTraits { };
template <typename R, typename O>
struct MemFuncTraits<R (O::*)()> {
  typedef R ReturnType;
  typedef O ObjectType;
};
template <typename R, typename O>
struct MemFuncTraits<R (O::*)() const> {
  typedef R ReturnType;
  typedef O ObjectType;
};
template <typename R, typename O, typename P1>
struct MemFuncTraits<R (O::*)(P1)> {
  typedef R ReturnType;
  typedef O ObjectType;
};
template <typename R, typename O, typename P1>
struct MemFuncTraits<R (O::*)(P1) const> {
  typedef R ReturnType;
  typedef O ObjectType;
};
template <typename MemFuncPtrType>
class PMFC {
public:
  typedef typename MemFuncTraits<MemFuncPtrType>::ObjectType ObjectType;
  typedef typename MemFuncTraits<MemFuncPtrType>::ReturnType ReturnType;
  typedef std::pair<ObjectType*, MemFuncPtrType> CallInfo;
  PMFC(const CallInfo& info) : _callinfo(info) { }
  // support for 0 parameter
  ReturnType operator()() const
    { return (_callinfo.first->*_callinfo.second)(); }
  // support for 1 parameter
  template <typename Param1Type>
    ReturnType operator()(Param1Type p1) const
      { return (_callinfo.first->*_callinfo.second)(p1); }
private:
  CallInfo _callinfo;
};
template <typename T>
class SmartPtrBase {
public:
  SmartPtrBase(T *p) : ptr(p) { }
```

```
  template <typename MemFuncPtrType>
    const PMFC<MemFuncPtrType> operator->*(MemFuncPtrType pmf) const
      {  return std::make_pair(ptr, pmf); }
private:
  T* ptr;  // dumb pointer
};
template <typename T>
class SP : public SmartPtrBase<T> {   // (1) 原文采用 private 继承，GCC 失败。注
public:
  SP(T *p) : SmartPtrBase<T>(p) { }
  using SmartPtrBase<T>::operator->*;  // (2) 在 G++ 中可编译但无作用
  // normal smart pointer functions would go here.
};
/*
I  注：
        原文 (1) 采用 private 继承，但是 GCC 不支援「以 using declaration 突破
private inheritance 的限制」（可却又允许 (2) 通过编译，怪！），造成 main()之中凡
使用 pw->* 即出现编译错误。只好在此改为 public。既然采用 public inheritance，(2)
的有无也就无关宏旨了。但是，《C++ Primer 中文版》p.981 讲「以 using declaration 突
破 private inheritance 的限制」，我以 981.cpp 试之，G++ 却又表现良好。为什麼？请
叁考：well-known bug in G++: "Using declarations in classes do not work!"见
Frequently Reported Bugs in GCC 2.95 (http://gcc.gnu.org/bugs.html)
*/
class Wombat {
public:
  int dig() { cout << "Digging..." << endl;  return 1; }
  int sleep() { cout << "Sleeping..." << endl;  return 5; }
  int eat() const { cout << "Eatting..." << endl;  return 7; }
  int move(int op) { cout << "Moving..." << endl;  return 9; }
};
typedef int(Wombat::*PWMF)();
typedef int(Wombat::*PWMFC)() const;
typedef int(Wombat::*PWMF1)(int);
void main()
{
  SP<Wombat> pw = new Wombat;
  PWMF pmf = &Wombat::dig;
  (pw->*pmf)();              // Digging...
  pmf = &Wombat::sleep;
  (pw->*pmf)();              // Sleeping...
  PWMFC pmfc = &Wombat::eat;
  (pw->*pmfc)();             // Eatting...
  PWMF1 pmf1 = &Wombat::move;
  (pw->*pmf1)(2);            // Eatting...
}
```

陈崴注 2：作者在文中提到 wombat 是一种澳洲特产的一种有袋动物，看起来有点像狗。但从像片来看并非如此。wombat 是夜行性动物，喜欢掘土，并不凶猛。下面是 wombat 的玉容：

作者也许想成另一种澳洲特产 dingo，那是野生而极似狗的凶猛动作，有攻击性，曾有小孩被这种动物咬死。