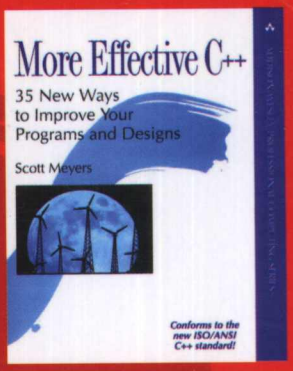
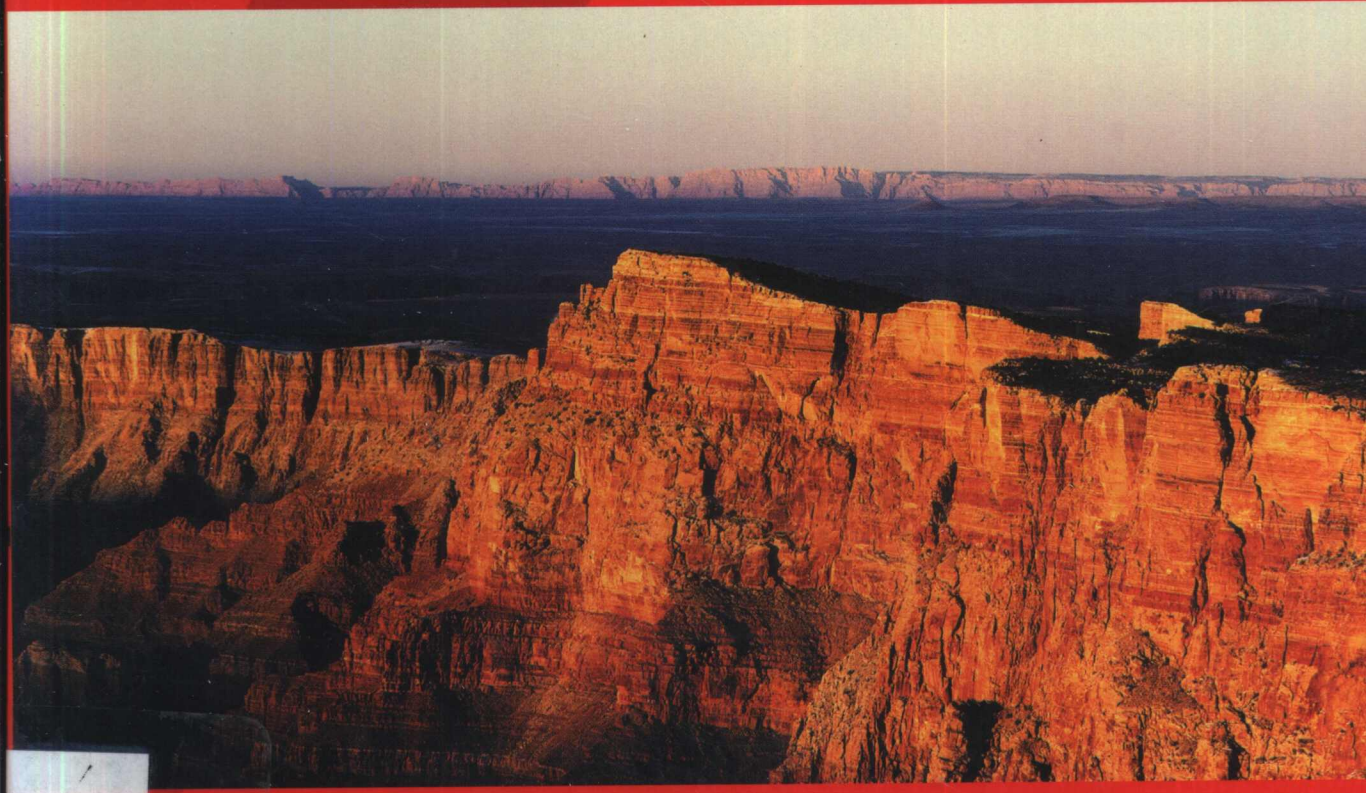
深入C++系列



More Effective C++ 中文版

Scott Meyers 著
侯 捷 译



Addison
Wesley

中国电力出版社
www.infopower.com.cn

More Effective C++

中文版

深入C++系列

More Effective C++
中文版

Scott Meyers 著
侯 捷 译

Addison
Wesley

中国电力出版社

More Effective C++: 35 New Ways to Improve Your Programs and Designs
(ISBN 0-201-63371-X)

Scott Meyers

图书在版编目（CIP）数据

More Effective C++中文版/（美）迈耶斯（Meyers,S.）著；侯捷译. —北京：中国电力出版社，2003.3

(深入 C++系列)

ISBN 7-5083-1486-7

Ⅰ.M… Ⅱ.①迈…②侯… Ⅲ.C 语言 - 程序设计 Ⅳ.TP312

中国版本图书馆 CIP 数据核字（2003）第 017799 号

责任编辑：常虹

丛 书 名： 深入C++系列

书　　名： More Effective C++中文版

编　　著： (美) 迈耶斯（Meyers,S.）

翻　　译： 侯 捷

出版发行： 中国电力出版社

地址：北京市三里河路6号 邮政编码：100044

电话：（010）88515918 传真：（010）88423191

印　　刷： 北京地矿印刷厂

开　　本： 787×1092 1/16　　**印　　张：** 20.75　　**字　　数：** 313 千字

书　　号： 7-5083-1486-7

版　　次： 2003年3月北京第一版

印　　次： 2003年3月第一次印刷

印　　数： 1～8000

定　　价： 50.00 元

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、语法背后的语义、语义背后的深层思维、深层思维背后的对象模型；C++ 的难学，还在于它提供了四种不同（相辅相成）的编程思维模型：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，又要回溯兼容，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，万千使用者前仆后继的动力是：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*），程序库大全有之（*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++ 相当基础的程序员而言，有着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++* 则是后起之秀。

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于 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，让我们了解问题背后的成因、最佳的解法、以及其他可能的牵扯。至于大范围主题，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！

我很高兴将以下两本优秀书籍，规划为一个系列，以郑重的形式呈现给您：

1. *Effective C++ 2/e*, by Scott Meyers, AW 1998
2. *More Effective C++*, by Scott Meyers, AW 1996

本书不但与英文版页页对译，保留索引，并加上译注、交叉索引[1]、读者服务[2]。

这套书将对于您的程序设计生涯带来重大帮助。翻译这套书籍的过程中，我感觉来自技术体会上的极大快乐。我祈盼（并相信）您在阅读此书时拥有同样的心情。

侯捷 2003/03/07 于台湾新竹
jjhou@jjhou.com
http://www.jjhou.com

本书保留大量简短易读之英文术语；时而中英并陈。以下用语特别请读者注意：

英文术语	本书译词	英文术语	本书译词
argument	自变量 （i.e. 实参）	instantiated	实体化、具现化
by reference	传址	library	程序库
by value	传值	resolve	决议
dereference	提领 （i.e. 解参考）	parameter	参数 （i.e. 形参）
evaluate	评估、核定	type	型别 （i.e. 类型）
instance	实体		

[1] *Effective C++ 2/e* 和 *More Effective C++* 之中译本，实际上是以 Scott Meyers 的另一个产品 *Effective C++ CD* 为本，不仅资料更新，同时亦将 CD 版中的两书交叉参考保留下来，可为读者带来旁征博引时的莫大帮助。

[2] 欢迎读者对本书所及主题提出讨论，并感谢读者对本书任何误失提出指正。来信请寄 jjhou@jjhou.com。勘误网站：http://www.jjhou.com（繁）http://jjhou.csdn.net（简）

致 谢

中文版 略

译注：藉此版面提醒读者，本书之中如果出现「条款 5」这样的参考指示，指的是本书条款 5。如果出现「条款 **E**5」这样的参考指示，**E** 是指 *Effective C++ 2/e*）

目 录

导读

Introduction

对 C++ 程序员而言，日子似乎有点过于急促。虽然只商业化不到 10 年，C++ 却俨然成为几乎所有主要计算环境的系统程序语言霸主。面临程序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，其所附带之标准程序库幅员广大，不仅涵盖 C 函数库，也使之相形见绌。这么一个大型程序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算法和数据结构的情况下，完成琳琅满目的各种复杂程序。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，它们所产生的代码品质也不断改善。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。商业化程序库几乎能够满足各个应用领域中的编程需求。

一旦语言进入成熟期，而我们对它的使用经验也愈来愈多，我们所需要的信息也就随之改变。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++ 程序员问的问题更高级：我如何能够设计出适应未来需求的软件？我如何能够改善代码的效率而不折损正确性和易用性？我如何能够实现语言未能直接支持的精巧机能？

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现 C++ 软件：让它行为更正确；面对异常时更稳健强固；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件更好。

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++ 程序设计社群所累积的智能。大部分条款以准则的形式呈现，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后果，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 专注于 exceptions（就像 Tom Cargill、Jack Reeves、HerbSutter 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 描述如何限制对象的个数或诞生地点，如何根据一个以上的对象型别产生出类似虚函数的东西，如何产生 smart pointers 等等。其他条款解决更广泛的题目。条款 16~24 专注于效率上的议题。不论哪一条款，提供的都是与其主题相关且意义重大的做法。在 *More Effective C++* 一书中你将学习到如何更实效更精锐地使用 C++。大部分 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信息而已。

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 classes（类）、保护层级（protection levels）、虚函数、非虚函数，我也假设你已通晓 templates 和 exceptions 背后的概念。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，我会进一步解释。

本书所谈的 C++

我在本书所谈、所用的 C++，是 ISO/ANSI 标准委员会于 1997 年 11 月完成的 C++ 国际标准最后草案（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) 支持能力之列。别担心，我认为对你而言惟一所谓「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，并大量集中于条款 9~15。如果你的编译器(s) 未能支持 exceptions，没什么大不了，这并不影响本书其他部分带给你的好处。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做法，并非就保证该语言特性已出现在目前的编译器上，或该实务做法已可应用于既有的开发环境上。

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。由于两者我都讨论，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仿真编译器(s) 尚未支持的语言特性。而当你决定将一些原本绕道而行的解决办法以新支持的语言特性取代时，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) 平台上发展代码。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，例如独家厂商才得出的某种语言新特性。如此特性往往实现不够精良（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代码，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。某些编译器支持它们，但如果你的编译器不支持，你可轻易以你所熟悉的其他性质来仿真它们。

第一个性质是 bool 型别，其值必为关键词 true 或 false。如果你的编译器尚未支持 bool，有两个方法可以仿真它。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 的不同函数加以重载（overloading）。缺点是，内建的「比较操作符（comparison operators）」如 ==, <, >=, 等等仍旧返回 ints。所以以下代码的行为不如我们所预期：

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

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

另一种做法是利用 typedef 来定义 bool，并以常量对象作为 true 和 false：

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

这种手法兼容于传统的 C/C++ 语义。使用这种仿真法的程序，在移植到一个支持有 bool 型别的编译器平台之后，行为并不会改变。缺点则是无法在函数重载（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，其实是四个转型操作符：static_cast, const_cast, dynamic_cast, 和 reinterpret_cast。如果你不熟悉这些转型操作符，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操作符。

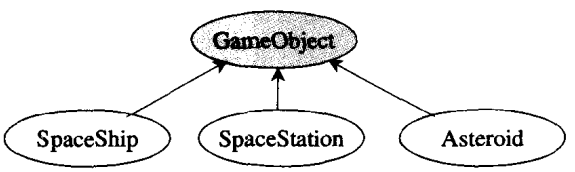
C++ 拥有比语言本身更丰富的东西。是的，C++ 还有一个伟大的标准程序库（见条款 E49）。我尽可能使用标准程序库所提供的 string 型别来取代 char* 指针，而且我也鼓励你这么做。string objects 并不比 char*-based 字符串难操作，它们的好处是可以免除你大部分的内存管理工作。而且如果发生 exception 的话（见条款 9 和 10），string objects 比较没有 memory leaks（内存泄漏）问题。实现良好的 string 型别甚至可和对应的 char* 比赛效率，而且可能会赢（条款 29 会告诉你其中故事）。如果你不打算使用标准的 string 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序库提供的数据结构。这些数据结构来自 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数据结构，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带 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算法和数据结构的程序库，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代码」之间可以选择，你应该让自己倾向使用 STL。记得代码的复用性吗？STL（以及标准程序库的其他组件）之中有许多代码是十分值得重复运用的。

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，我会明白地指出。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被广泛接受的箭头画法：从 derived classes 画往 base classes，而且我很高兴事情终能归于一统。此类示意图中，抽象类（abstract classes，例如上图的 `GameObject`）被我加上阴影而具象类（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」是指其声明时的型别，「动态型别」则由它们实际所指的对象来决定。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*，
                                 // 至于其动态型别如今是 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后的基本原理，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」的函数，我可能会这样声明：

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

这使我得以写出这样的代码：

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

在调用 operator== 的过程中，r1 位于 "==" 左侧，被绑定于 lhs，r2 位于 "==" 右侧，被绑定于 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 的支持（在此性质中，dynamic_cast 是最常被使用的一个零组件）。

当你分配内存而没有释放它，你就有了 memory leak（内存泄漏）问题。Memory leaks 在 C 和 C++ 中都有，但是在 C++ 中，memory leaks 所泄漏的还不只是内存，因为 C++ 会在对象被产生时，自动调用 constructors，而 constructors 本身可能亦配有资源（resources）。举个例子，考虑以下代码：

```
class Widget { ... };          // 某个 class — 它是什么并不重要。
Widget *pw = new Widget;       // 动态分配一个 Widget 对象。
...                            // 假设 pw 一直未被删除（deleted）。
```

这段代码会泄漏内存，因为 pw 所指的 Widget 对象从未被删除。如果 Widget constructor 分配了其他资源（例如 file descriptors, semaphores, window handles,

database locks），这些资源原本应该在 `Widget` 对象被销毁时释放，现在也像内存一样都泄漏掉了。为了强调在 C++ 中 memory leaks 往往也会泄漏其他资源，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 函数。并不是我不喜欢 inlining，事实上我相信 inline 函数是 C++ 的一项重要性质。然而决定一个函数是否应被 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 函数，并不意味我认为把它声明为 `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。这样的性质被明列于语言的最后撤除名单，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，会尽力保存向下兼容性，所以那些被撤除的性质大约还会存活好多年。

所谓 **client**，是指你所写的代码的客户。或许是某些人（程序员），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、最后期限、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 的代码，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件，你又何必写它呢？你会发现我很在意如何让 clients 更轻松，通常这会导致你的行事更困难，因为好的软件「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