传世经典书丛
Eternal Classics

More Effective C++ 中文版

35个改善编程与设计的有效方法

More Effective C++:
35 New Ways to Improve Your Programs and Designs

[美] Scott Meyers 著
侯 捷 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

继 *Effective* C++之后，Scott Meyers 于 1996 推出这本“续集”。条款变得比较少，页数倒是多了一些，原因是这次选材比“第一集”更高阶，尤其是第 5 章。Meyers 将此章命名为技术（techniques），并明白告诉你，其中都是一些 patterns，例如 virtual constructors，smart pointers，reference counting，proxy classes，double dispatching……这一章的每个条款篇幅都达 15~30 页之多，实在让人有“山重水复疑无路，柳暗花明又一村”之叹。

虽然出版年代稍嫌久远，但本书并没有第 2 版，原因是当其出版之时（1996），C++ Standard 已经几乎定案，本书即依当时的标准草案而写，其与现今的C++标准规范几乎相同。而且可能变化的几个弹性之处，Meyers也都有所说明与提示。读者可以登录作者提供的网址，看看上下两集的勘误与讨论（数量之多，令人惊恐。幸好多是技术讨论或文字斟酌，并没有什么重大误失）。

Authorized translation from the English language edition, entitled More Effective C++: 35 New Ways to Improve Your Programs and Designs, 1st Edition, 020163371X by Scott Meyers, published by Pearson Education, Inc., publishing as Addison Wesley Professional, Copyright©1996 Pearson Education Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD., and PUBLISHING HOUSE OF ELECTORNICS INDUSTRY Copyright ©2010

本书简体中文版专有出版权由 Pearson Education 培生教育出版亚洲有限公司授予电子工业出版社。未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。

本书简体中文版贴有 Pearson Education 培生教育出版集团激光防伪标签，无标签者不得销售。

版权贸易合同登记号　图字：01-2010-7892

图书在版编目（CIP）数据

More Effective C++：35 个改善编程与设计的有效方法/（美）梅耶（Meyers,S.）著；侯捷译. 北京：电子工业出版社，2011.1
（传世经典书丛）
书名原文：More Effective C++:35 New Ways to Improve Your Programs and Designs
ISBN 978-7-121-12570-6

Ⅰ. ①M… Ⅱ. ①梅… ②侯… Ⅲ. ①C 语言－程序设计 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字（2010）第 247371 号

策划编辑：张春雨
责任编辑：付　睿　　李云静
印　　刷：北京中新伟业印刷有限公司
装　　订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　　1/16　　印张：21　　字数：500 千字
版　　次：2011 年 1 月第 1 版
印　　次：2015 年 2 月第 9 次印刷
定　　价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。
服务热线：（010）88258888。

More Effective C++ 的荣耀：

这是一本多方面发人深省的 C++ 书籍：不论在你偶尔用到的语言特性上，或是在你自以为十分熟悉的语言特性上。只有深刻了解 C++ 编译器如何解释你的代码，你才有可能用 C++ 语言写出健壮的软件。本书是协助你获得此等层级的了解过程中，一份极具价值的资源。读过本书之后，我感觉像是浏览了 C++ 编程大师所检阅过的代码，并获得许多极具价值的洞见。

——Fred Wild, Vice President of Technology,
Advantage Software Technologies

本书内含大量重要的技术，这些技术是撰写优良 C++ 程序所不可或缺的。本书解释如何设计和实现这些观念，以及潜伏在其他某些替代方案中的陷阱。本书亦含晚些加入的 C++特性的详细说明。任何人如果想要好好地运用这些新特性，最好买一本并且放在随手可得之处，以备查阅。

——Christopher J. Van Wyk, Professor
Mathematics and Computer Science, Drew University

这是一本具备工业强度的最佳书籍。对于已经阅读过 *Effective C++* 的人，这是完美的续集。

——Eric Nagler, C++ Instructor and Author,
University of California Santa Cruz Extension

More Effective C++ 是一本无微不至且很有价值的书籍，是 Scott 第一本书 *Effective C++* 的续集。我相信每一位专业的 C++ 软件开发人员都应该读过并记忆 *Effective C++* 和 *More Effective C++* 两本书内的各种招式，以及其中重要（而且有时候不可思议）的语言的方方面面。我强烈推荐这两本书给软件开发人员、测试人员、管理人员……，每个人都可以从 Scott 专家级的知识与卓越的表达能力中获益。

——Steve Burkett, Software Consultant

For Clancy,

my favorite enemy within.

献给每一位对 C++/OOP 有所渴望的人

正确的观念 重于一切

— 侯 捷 —

译序

C++ 是一门难学易用的语言！

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 4 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，万千使用者前赴后继的动力是：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。

这类书籍的一个共同特色是轻薄短小，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。它们不但开扩读者的视野，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
http://www.wzbook.org

[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，不仅数据更新，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失误提出指正。来信请寄 jjhou@jjhou.com。勘误网站：http://www.jjhou.com（繁），http://jjhou.csdn.net（简）

本书保留大量简短易读之英文术语，时而中英并陈。以下用语请读者特别注意：

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	实例化、具现化
by reference	传址	library	程序库
by value	传值	resolve	决议
dereference	解引（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	实例		

译注：借此版面提醒读者，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

译序（侯捷） ix

导读（Introduction） 001

基础议题（Basics） 009

条款 1：仔细区别 pointers 和 references 009
Distinguish between pointers and references.

条款 2：最好使用 C++ 转型操作符 012
Prefer C++-style casts.

条款 3：绝对不要以多态（polymorphically）方式处理数组 016
Never treat arrays polymorphically.

条款 4：非必要不提供 default constructor 019
Avoid gratuitous default constructors.

操作符（Operators） 024

条款 5：对定制的“类型转换函数”保持警觉 024
Be wary of user-defined conversion functions.

条款 6：区别 increment/decrement 操作符的
前置（prefix）和后置（postfix）形式 031
Distinguish between prefix and postfix forms of increment
and decrement operators.

条款 7：千万不要重载&&，||和，操作符 035
Never overload &&, ||, or , .

条款 8：了解各种不同意义的 new 和 delete 038
Understand the different meanings of new and delete

异常（Exceptions） 044

条款 9：利用 destructors 避免泄漏资源 045
Use destructors to prevent resource leaks.

条款 10：在 constructors 内阻止资源泄漏（resource leak） 050
Prevent resource leaks in constructors.

条款 11：禁止异常（exceptions）流出 destructors 之外 058
Prevent exceptions from leaving destructors.

条款 12：了解“抛出一个 exception”与“传递一个参数”或“调用一个虚函数”之间的差异 061
Understand how throwing an exception differs from passing a parameter or calling a virtual function.

条款 13：以 by reference 方式捕捉 exceptions 068
Catch exceptions by reference.

条款 14：明智运用 exception specifications 072
Use exception specifications judiciously.

条款 15：了解异常处理（exception handling）的成本 078
Understand the costs of exception handling.

效率（Efficiency） 081

条款 16：谨记 80-20 法则 082
Remember the 80-20 rule.

条款 17：考虑使用 lazy evaluation（缓式评估） 085
Consider using lazy evaluation.

条款 18：分期摊还预期的计算成本 093
Amortize the cost of expected computations.

条款 19：了解临时对象的来源 098
Understand the origin of temporary objects.

条款 20：协助完成“返回值优化（RVO）” 101
Facilitate the return value optimization.

条款 21：利用重载技术（overload）避免隐式类型转换（implict type conversions） 105
Overload to avoid implicit type conversions.

条款 22：考虑以操作符复合形式（op=）取代其独身形式（op） 107
Consider using *op=* instead of stand-alone *op*.

条款 23：考虑使用其他程序库 110
Consider alternative libraries.

条款 24：了解 virtual functions、multiple inheritance、virtual base classes、runtime type identification 的成本 113
Understand the costs of virtual functions, multiple inheritance, virtual base classes, and RTTI.

技术（Techniques, Idioms, Patterns） 123

条款 25：将 constructor 和 non-member functions 虚化 123
Virtualizing constructors and non-member functions.

条款 26：限制某个 class 所能产生的对象数量 130
Limiting the number of objects of a class.

条款 27：要求（或禁止）对象产生于 heap 之中 145
Requiring or prohibiting heap-based objects.

条款 28：Smart Pointers（智能指针） 159

条款 29：Reference counting（引用计数） 183

条款 30：Proxy classes（替身类、代理类） 213

条款 31：让函数根据一个以上的对象类型来决定如何虚化 228
Making functions virtual with respect to more than one object.

杂项讨论（Miscellany） 252

条款 32：在未来时态下发展程序 252
Program in the future tense.

条款 33：将非尾端类（non-leaf classes）设计为抽象类（abstract classes） 258
Make non-leaf classes abstract.

条款 34：如何在同一个程序中结合 C++ 和 C 270
Understand how to combine C++ and C in the same program.

条款 35：让自己习惯于标准 C++ 语言 277
Familiarize yourself with the language standard.

推荐读物（Recommended Reading） 285

auto_ptr 实现代码 291

索引（一）（General Index） 295

索引（二）（Index of Example Classes，Functions，and Templates） 313

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，其所附带的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，它们所产生的代码质量也不断改善。C++ 开发工具和开发环境愈来愈丰富，威力愈来愈强大，健壮（robust）的程度也愈来愈高。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，我们对它的使用经验也就愈来愈多，我们所需要的信息也就随之改变了。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不能直接支持的精巧功能？

这本书中我要回答这些问题，以及其他许多类似问题。

本书将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，面对异常时更健壮、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”开发环境中运作得更好，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++ 程序设计社区所积累的智能。大部分条款以准则的形式呈现，随附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，特别是你可能少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，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 描述如何限制对象的个数或诞生地点，如何根据一个以上的对象类型产生出类似虚函数的东西，如何产生 smart pointers 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。不论哪一个条款，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。

这种处理方式意味着，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，我认为对你而言唯一所谓的“新”特性，应该只有 templates，而 templates 如今几乎已是各家编译器的必备功能。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，即使你不需要用到 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，我便将两者都加以讨论，虽然我其实更重视实务。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模拟编译器（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释。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。如此特性往往实现得不够精良（臭虫多，要不就表现得迟缓，或是两者兼具），而且 C++ 社区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生可靠的代码，恐怕还是步步为营（并且能够与人合作）的好。

本书用了两个你可能不太熟悉的 C++ 性质，它们都是晚些才加入 C++ 标准之中的。某些编译器支持它们，但如果你的编译器不支持，你可以轻易地用你所熟悉的其他性质来模拟它们。

第一个性质是 `bool` 类型，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` 的不同函数加以重载（overloading）。缺点是，内建的“比较操作符（comparison operators）”如 ==，<，>=，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;
```

这种手法与传统的 C/C++ 语义兼容。使用这种仿真法的程序，在移植到一个支持有 bool 类型的编译器平台之后，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 4 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。

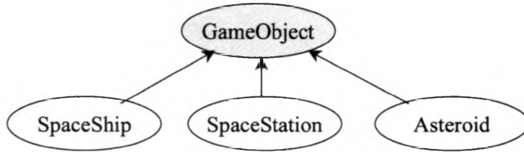
C++ 拥有比语言本身更丰富的东西。是的，C++ 还有一个伟大的标准程序库（见条款 E49）。我尽可能使用标准程序库所提供的 string 类型来取代 char* 指针，而且我也鼓励你这么做。string objects 并不比 char*-based 字符串难操作，它们的好处是可以免除你大部分的内存管理工作。而且如果发生 exception 的话（见条款 9 和 10），string objects 比较不会出现 memory leaks（内存泄漏）问题。实现良好的 string 类型甚至可和对应的 char* 比赛效率，而且可能会赢（条款 29 会告诉你其中的故事）。如果你不打算使用标准的 string 类型，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配合使用。

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，我会明确地指出。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`）被我加上阴影而具体类（concrete classes，例如上图的 `SpaceShip`）未加阴影。

Inheritance（继承）机制会引发“pointers（或 references）拥有两个不同的类型”的议题，两个类型分别是静态类型（static type）和动态类型（dynamic type）。Pointer 或 reference 的“静态类型”是指其声明时的类型，“动态类型”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 的静态类型是 GameObject*，
  new SpaceShip;               // 动态类型是 SpaceShip*。
Asteroid *pa = new Asteroid;   // pa 的静态类型是 Asteroid*，
                               // 动态类型也是 Asteroid*。
pgo = pa;                 //pgo 的静态类型仍然（永远）是 GameObject*，
                          // 至于其动态类型如今是 Asteroid*。
```

```
GameObject& rgo = *pa;          // rgo 的静态类型是 GameObject，
                                // 动态类型是 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“用来比较两个 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，而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++ 性质已明确地被标准委员会排除。这样的性质被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。这本书中我会找出被撤除的性质，并说明其取代者。你应该避免使用被撤除的性质，但是过度在意倒也不必，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），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 templates 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（实例）之间的差异。举个例子，如果 Array 是个 class template，有个类型参数 T，我可能会以 Array 代表此 template 的某个特定实例（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，有个类型参数 T，我可能会以 swap 而非 swap<T> 表示其实例。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 实例时加上 template 参数。

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，或任何其他问题——请告诉我，我会试着在本书重印时修正。如果你是第一位告诉我的人，我会很高兴将你的大名记录到本书致谢文（acknowledgments）内。如果你有改善建议，我也非常欢迎。

我将继续收集 C++ 程序设计的实效准则。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.
```

或者你也可以发送电子邮件到 mec++@awl.com。

我整理了一份本书第一次印刷以来的修订记录，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及与本书相关的其他信息。你也可以通过 anonymous FTP，从 ftp.awl.com 的 cp/mec++ 目录中取得它。如果你希望拥有这份数据，但又无法上网，请寄申请函到上述地址，我会邮寄一份给你。

这篇序文够长的了，让我们开始正题吧。

基础议题

Basics

基础议题。是的，pointers（指针）、references（引用）、casts（类型转换）、arrays（数组）、constructors（构造函数）——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分特性，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分，但有时候它们还是会令你吃惊。特别是对那些从 C 转战到 C++ 的程序员，因为 references，dynamic casts，default constructors 及其他 non-C 性质背后的观念，往往带有一股黝黯阴郁的色彩。

这一章描述 pointers 和 references 的差异，并告诉你它们的适当使用时机。本章介绍新的 C++ 转型（casts）语法，并解释为什么新式转型法比旧式的 C 转型法优越。本章也检验了 C 的数组概念及 C++ 的多态（polymorphism）概念，并说明为什么将这两者混用是不智之举。最后，本章讨论 default constructors（默认构造函数）的正方和反方意见，并提出一些建议做法，让你回避语言的束缚（因为在你不需要 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，你将向着一个很好的目标迈进：你所编写的软件可以清楚而正确地表现出你的设计意图。

条款 1：仔细区别 pointers 和 references

Pointers 和 references 看起来很不一样（pointers 使用“*”和“->”操作符，references 则使用“.”），但它们似乎做类似的事情。不论 pointers 或是 references 都使你得间接参考其他对象。那么，何时使用哪一个？你心中可有一把尺？

首先你必须认知一点，没有所谓的 null reference。一个 reference 必须总代表

某个对象。所以如果你有一个变量，其目的是用来指向（代表）另一个对象，但是也有可能它不指向（代表）任何对象，那么你应该使用 pointer，因为你可以将 pointer 设为 null。换个角度看，如果这个变量总是必须代表一个对象，也就是说如果你的设计并不允许这个变量为 null，那么你应该使用 reference。

“但是等等”你说，“下面这样的东西，底层意义是什么呢？”

```
char *pc = 0;          // 将 pointer 设定为 null。
char& rc = *pc;        // 让 reference 代表 null pointer 的解引值。
```

哦，这是有害的行为，其结果不可预期（C++ 对此没有定义），编译器可以产生任何可能的输出，而写出这种代码的人，应该与大众隔离，直到他们允诺不再有类似行为。如果你在你的软件中还需担心这类事情，我建议你还是完全不要使用 references 的好，要不就是另请一个比较高明的程序员来负责这类事情。从现在起，我们将永远不再考虑“reference 成为 null”的可能性。

由于 reference 一定得代表某个对象，C++ 因此要求 references 必须有初值：

```
string& rs;                // 错误！references 必须被初始化。
string s("xyzzy");
string& rs = s;            // 没问题，rs 指向 s。
```

但是 pointers 就没有这样的限制：

```
string *ps;             // 未初始化的指针，有效，但风险高。
```

“没有所谓的 null reference”这个事实意味使用 references 可能会比使用 pointers 更富效率。这是因为使用 reference 之前不需要测试其有效性：

```
void printDouble(const double& rd)
{
    cout << rd;     // 不需要测试 rd，它一定代表某个 double。
}
```

如果使用 pointers，通常就得测试它是否为 null：

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

Pointers 和 references 之间的另一个重要差异就是，pointers 可以被重新赋值，指向另一个对象，reference 却总是指向（代表）它最初获得的那个对象：

```
string s1("Nancy");
string s2("Clancy");

string& rs = s1;          // rs 代表 s1。
string *ps = &s1;         // ps 指向 s1。
rs = s2;                  // rs 仍然代表 s1，
                          // 但是 s1 的值现在变成了“Clancy”。
ps = &s2;                 // ps 现在指向 s2，
                          // s1 没有变化。
```

一般而言，当你需要考虑“不指向任何对象”的可能性时，或是考虑“在不同时间指向不同对象”的能力时，你就应该采用 pointer。前一种情况你可以将 pointer 设为 null，后一种情况你可以改变 pointer 所指对象。而当你确定“总是会代表某个对象”，而且“一旦代表了该对象就不能够再改变”，那么你应该选用 reference。

还有其他情况也需要使用 reference，例如当你实现某些操作符的时候。最常见的例子就是 operator[]。这个操作符很特别地必须返回某种“能够被当做 assignment 赋值对象”的东西：

```
vector<int> v(10);    // 产生一个 int vector，大小为 10。
                      // vector 是 C++ 标准程序库（见条款 35）
                      // 提供的一个 template。
v[5] = 10;            // assignment 的赋值对象是 operator[] 的返回值。
```

如果 operator[] 返回 pointer，上述最后一个语句就必须写成这样子：

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

但这使 v 看起来好像是个以指针形成的 vector，事实上它不是。为了这个因素，你应该总是令 operator[] 返回一个 reference。条款 30 有一个例外，十分有趣。

因此，让我做下结论：当你知道你需要指向某个东西，而且绝不会改变指向其他东西，或是当你实现一个操作符而其语法需求无法由 pointers 达成，你就应该选择 references。任何其他时候，请采用 pointers。

条款 2：最好使用 C++ 转型操作符

想想低阶转型动作。它几乎像 goto 一样被视为程序设计上的“贱民”。尽管如此，它却仍能够苟延残喘，因为当某种情况愈来愈糟，转型可能是必要的。是的，当某种情况愈来愈糟，转型是必要的！

不过，旧式的 C 转型方式并非是唯一选择。它几乎允许你将任何类型转换为任何其他类型，这是十分拙劣的。如果每次转型都能够更精确地指明意图，则更好。举个例子，将一个 pointer-to-const-object 转型为一个 pointer-to-non-const-object（也就是说只改变对象的常量性），和将一个 pointer-to-base-class-object 转型为一个 pointer-to-derived-class-object（也就是完全改变了一个对象的类型），其间有很大的差异。传统的 C 转型动作对此并无区分（这应该不会造成你的惊讶，因为 C 式转型是为 C 设计的，不是为了 C++）。

旧式转型的第二个问题是它们难以辨识。旧式转型的语法结构是由一对小括号加上一个对象名称（标识符）组成，而小括号和对象名称在 C++ 的任何地方都有可能被使用。因此，我们简直无法回答最基本的转型相关问题“这个程序中有使用任何转型动作吗？”。因为人们很可能对转型动作视而不见，而诸如 grep 之类的工具又无法区分语法上极类似的一些非转型写法。

为解决 C 旧式转型的缺点，C++ 导入 4 个新的转型操作符（cast operators）：static_cast，const_cast，dynamic_cast 和 reinterpret_cast。对大部分使用目的而言，面对这些操作符你唯一需要知道的便是，过去习惯的写码形式：

```
(type) expression
```

现在应该改为这样：

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

举个例子，假设你想要将一个 int 转型为一个 double，以强迫一个整数表达式导出一个浮点数值来。采用 C 旧式转型，可以这么做：

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

如果采用新的 C++ 转型法，应该这么写：

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

这种形式十分容易被辨识出来，不论是对人类或是对工具程序而言。

static_cast 基本上拥有与 C 旧式转型相同的威力与意义，以及相同的限制。例如，你不能够利用 static_cast 将一个 struct 转型为 int，或将一个 double 转型为 pointer；这些都是 C 旧式转型动作原本就不可以完成的任务。static_cast 甚至不能够移除表达式的常量性（constness），因为有一个新式转型操作符 const_cast 专司此职。

其他新式 C++ 转型操作符适用于更集中（范围更狭窄）的目的。const_cast 用来改变表达式中的常量性（constness）或变易性（volatileness）。使用 const_cast，便是对人类（以及编译器）强调，通过这个转型操作符，你唯一打算改变的是某物的常量性或变易性。这项意愿将由编译器贯彻执行。如果你将 const_cast 应用于上述以外的用途，那么转型动作会被拒绝。下面是个例子：

```
class Widget { ... };
class SpecialWidget: public Widget { ... };
void update(SpecialWidget *psw);
SpecialWidget sw;                   // sw 是个 non-const 对象，
const SpecialWidget& csw = sw;      // csw 却是一个代表 sw 的 reference，
                                    // 并视之为一个 const 对象。

update(&csw);           // 错误！不能将 const SpecialWidget*
                        // 传给一个需要 SpecialWidget* 的函数。

update(const_cast<SpecialWidget*>(&csw));
                        // 可！&csw 的常量性被去除了。也因此，
                        // csw（亦即 sw）在此函数中可被更改。

update((SpecialWidget*)&csw);
                        // 情况同上，但使用的是较难辨识
                        // 的 C 旧式转型语法。

Widget *pw = new SpecialWidget;
update(pw);             // 错误！pw 的类型是 Widget*，但
                        // update() 需要的却是 SpecialWidget*。

update(const_cast<SpecialWidget*>(pw));
                        // 错误！const_cast 只能用来影响
                        // 常量性或变易性，无法进行继承体系
                        // 的向下转型（cast down）动作。
```

显然，const_cast 最常见的用途就是将某个对象的常量性去除掉。

第二个特殊化的转型操作符是 dynamic_cast，用来执行继承体系中“安全的向下转型或跨系转型动作”。也就是说你可以利用 dynamic_cast，将“指向 base class objects 的 pointers 或 references”转型为“指向 derived（或 sibling base）class objects 的 pointers 或 references”，并得知转型是否成功[1]。如果转型失败，会以一个 null 指针（当转型对象是指针）或一个 exception（当转型对象是 reference）表现出来：

```
Widget *pw;
...
update(dynamic_cast<SpecialWidget*>(pw));
                         // 很好，传给 update() 一个指针，指向
                         // pw 所指的 SpecialWidget——如果 pw
                         // 真的指向这样的东西；否则传过去的
                         // 将是一个 null 指针。
void updateViaRef(SpecialWidget& rsw);
updateViaRef(dynamic_cast<SpecialWidget&>(*pw));
                         // 很好，传给 updateViaRef() 的是
                         // pw 所指的 SpecialWidget——如果
                         // pw 真的指向这样的东西；否则
                         // 抛出一个 exception。
```

dynamic_cast 只能用来协助你巡航于继承体系之中。它无法应用在缺乏虚函数（请看条款 24）的类型身上，也不能改变类型的常量性（constness）：

```
int firstNumber, secondNumber;
...
double result =
dynamic_cast<double>(firstNumber)/secondNumber;
                         // 错误！未涉及继承机制。
const SpecialWidget sw;
...
update(dynamic_cast<SpecialWidget*>(&sw));
                         // 错误！dynamic_cast 不能改变常量性。
```

如果你想为一个不涉及继承机制的类型执行转型动作，可使用 static_cast；要改变常量性（constness），则必须使用 const_cast。

最后一个转型操作符是 reinterpret_cast。这个操作符的转换结果几乎总是与编译平台息息相关。所以 reinterpret_casts 不具移植性。

---

1 dynamic_cast 的第二个（与第一个不相干）用途是找出被某对象占用的内存的起始点。我将在条款 27 解释这项能力。

reinterpret_cast 的最常用用途是转换“函数指针”类型。假设有一个数组，存储的都是函数指针，有特定的类型：

```
typedef void (*FuncPtr)();      // FuncPtr 是个指针，指向某个函数。
                                // 后者无须任何自变量，返回值为 void。
FuncPtr funcPtrArray[10];       // funcPtrArray 是个数组，
                                // 内有 10 个 FuncPtrs。
```

假设由于某种原因，你希望将以下函数的一个指针放进 funcPtrArray 中：

```
int doSomething();
```

如果没有转型，不可能办到这一点，因为 doSomething 的类型与 funcPtrArray 所能接受的不同。funcPtrArray 内各函数指针所指函数的返回值是 void，但 doSomething 的返回值却是 int：

```
funcPtrArray[0] = &doSomething;     // 错误！类型不符。
```

使用 reinterpret_cast，可以强迫编译器了解你的意图。

```
funcPtrArray[0] =                   // 这样便可通过编译。
  reinterpret_cast<FuncPtr>(&doSomething);
```

函数指针的转型动作，并不具移植性（C++ 不保证所有的函数指针都能以此方式重新呈现），某些情况下这样的转型可能会导致不正确的结果（见条款 31），所以你应该尽量避免将函数指针转型，除非你已走投无路，像是被逼到墙角，而且有一把刀子抵住你的喉咙。一把锐利的刀子，非常锐利的刀子。

如果你的编译器尚未支持这些新式转型动作，你可以使用传统转型方式来取代 static_cast，const_cast 和 reinterpret_cast。甚至可以利用宏（macros）来仿真这些新语法。

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

上述新语法的使用方式如下：

```
double result = static_cast(double, firstNumber)/secondNumber;
update(const_cast(SpecialWidget*, &sw));
funcPtrArray[0] = reinterpret_cast(FuncPtr, &doSomething);
```

这些近似法当然不像其本尊那么安全，但如果你现在就使用它们，一旦你的编译器开始支持新式转型，程序升级的过程便可简化。

至于 dynamic_cast，没有什么简单方法可以模拟其行为，不过许多程序库提供了一些函数，用来执行继承体系下的安全转型动作。如果你手上没有这些函数，而却必须执行这类转型，你也可以回头使用旧式的 C 转型语法，但它们不可能告诉你转型是否成功。当然，你也可以定义一个宏，看起来像 dynamic_cast，就像你为其他转型操作符所做的那样：

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

这个近似法并非执行真正的 dynamic_cast，所以它无法告诉你转型是否成功。

我知道，我知道，这些新式转型操作符看起来又臭又长。如果你实在看它们不顺眼，值得安慰的是 C 旧式转型语法仍然可继续使用。然而这么一来也就丧失了新式转型操作符所提供的严谨意义与易辨识度。如果你在程序中使用新式转型法，比较容易被解析（不论是对人类还是对工具而言），编译器也因此得以诊断转型错误（那是旧式转型法侦测不到的）。这些都是促使我们舍弃 C 旧式转型语法的重要因素。至于可能的第三个因素是：让转型动作既丑陋又不易键入（typing），或许未尝不是件好事。

条款 3：绝对不要以多态（polymorphically）方式处理数组

继承（inheritance）的最重要性质之一就是：你可以通过“指向 base class objects”的 pointers 或 references，来操作 derived class objects。如此的 pointers 和 references，我们说其行为是多态的（polymorphically）——犹如它们有多重类型似的。C++ 也允许你通过 base class 的 pointers 和 references 来操作“derived class objects 所形成的数组”。但这一点也不值得沾沾自喜，因为它几乎绝不会如你所预期般地运作。

举个例子，假设你有一个 class BST（意思是 binary search tree）及一个继承自 BST 的 class BalancedBST：

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

在一个真正具规模的程序中，这样的 classes 可能会被设计为 templates，不过这不是此处重点；如果加上 template 各种语法，反而使程序更难阅读。针对目前的讨论，我假设 BST 和 BalancedBST 都只内含 ints。

现在考虑有个函数，用来打印 BSTs 数组中的每一个 BST 的内容：

```
void printBSTArray(ostream& s, const BST array[], int numElements)
{
  for (int i = 0; i < numElements; ++i) {
    s << array[i];                // 假设 BST objects 有一个
  }                               // operator<< 可用。
}
```

当你将一个由 BST 对象组成的数组传给此函数，没问题：

```
BST BSTArray[10];
...
printBSTArray(cout, BSTArray, 10);        // 运行良好。
```

然而如果你将一个 BalancedBST 对象所组成的数组交给 printBSTArray 函数，会发生什么事：

```
BalancedBST bBSTArray[10];
...
printBSTArray(cout, bBSTArray, 10);        // 可以正常运行吗？
```

你的编译器会毫无怨言地接受它，但是看看这个循环（就是稍早出现的那一个）。

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

array[i] 其实是一个“指针算术表达式”的简写：它代表的其实是 *(array+i)。我们知道，array 是个指针，指向数组起始处。array 所指内存和 array+i 所指内存两者相距多远？答案是 i*sizeof(数组中的对象)，因为 array[0] 和 array[i] 之间有 i 个对象。为了让编译器所产生的代码能够正确走访整个数组，编译器必须有能力决定数组中的对象大小。很容易呀，参数 array 不是被声明为“类型为 BST”的数组吗？所以数组中的每个元素必然都是 BST 对象，所以 array 和 array+i 之间的距离一定是 i*sizeof(BST)。

至少你的编译器是这么想的。但如果你交给 printBSTArray 函数一个由 BalancedBST 对象组成的数组，你的编译器就会被误导。这种情况下它仍假设数组中每一元素的大小是 BST 的大小，但其实每一元素的大小是 BalancedBST 的大小。由于 derived classes 通常比其 base classes 有更多的 data members，所以 derived class objects 通常都比其 base class objects 来得大。因此，我们可以合理地

预期一个 BalancedBST object 比一个 BST object 大。如果是这样，编译器为 printBSTArray 函数所产生的指针算术表达式，对于 BalancedBST objects 所组成的数组而言就是错误的。至于会发生什么结果，不可预期。无论如何，结果不会令人愉快。

如果你尝试通过一个 base class 指针，删除一个由 derived class objects 组成的数组，那么上述问题还会以另一种不同面貌出现。下面是你可能做出的错误尝试：

```
// 删除一个数组，但是首先记录一个有关此删除动作的消息。
void deleteArray(ostream& logStream, BST array[])
{
  logStream << "Deleting array at address "
            << static_cast<void*>(array) << '\n';
  delete [] array;
}

BalancedBST *balTreeArray =         // 产生一个 BalancedBST 数组。
  new BalancedBST[50];
...
deleteArray(cout, balTreeArray);    // 记录此删除动作。
```

虽然你没有看到，但其中一样有“指针算术表达式”的存在。是的，当数组被删除，数组中每一个元素的 destructor 都必须被调用（见条款 8），所以当编译器看到这样的句子：

```
delete [] array;
```

必须产生出类似这样的代码：

```
// 将 *array 中的对象以其构造顺序的相反顺序加以析构。
for (int i = the number of elements in the array - 1; i >= 0; --i)
{
  array[i].BST::~BST();          // 调用 array[i] 的 destructor。
}
```

如果你这么写，便是一个行为错误的循环。编译器如果产生类似代码，当然同样是个行为错误的循环。C++语言规范中说，通过 base class 指针删除一个由 derived classes objects 构成的数组，其结果未定义。我们知道所谓“未定义”的意思就是：执行之后会产生苦恼。简单地说，多态（polymorphism）和指针算术不能混用。数组对象几乎总是会涉及指针的算术运算，所以数组和多态不要混用。

注意，如果你避免让一个具体类（如本例之 BalancedBST）继承自另一个具体

类（如本例之 BST），你就不太能够犯“以多态方式来处理数组”的错误。如条款 33 所说，设计你的软件使“具体类不要继承自另一个具体类”，可以带来许多好处。我鼓励你翻开条款 33，好好看看完整内容。

条款 4：非必要不提供 default constructor

所谓 default constructor（也就是说不给任何自变量就可调用者）是 C++ 一种“无中生有”的方式。Constructors 用来将对象初始化，所以 default constructors 的意思是在没有任何外来信息的情况将对象初始化。有时候可以想象，例如，数值之类的对象，可以被合理地初始化为 0 或一个无意义值。其他诸如指针之类的对象（条款 28）亦可被合理地初始化为 null 或无意义值。数据结构如 linked lists，hash tables，maps 等，可被初始化为空容器。

但是并非所有对象都落入这样的分类。有许多对象，如果没有外来信息，就没办法执行一个完全的初始化动作。例如，一个用来表现通信簿字段的 class，如果没有获得外界指定的人名，产生出来的对象将毫无意义。在某些公司，所有仪器设备都必须贴上一个识别号码。为这种用途（用以模拟出仪器设备）而产生的对象，如果其中没有供应适当的 ID 号码，将毫无意义。

在一个完美的世界中，凡可以“合理地从无到有生成对象”的 classes，都应该内含 default constructors，而“必须有某些外来信息才能生成对象”的 classes，则不必拥有 default constructors。但我们的世界毕竟不是完美的世界，所以我们必须纳入其他考虑。更明确地说，如果 class 缺乏一个 default constructor，当你使用这个 class 时便会有某些限制。

考虑下面这个针对公司仪器而设计的 class，在其中，仪器识别码是一定得有的一个 constructor 自变量：

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

由于 EquipmentPiece 缺乏 default constructor，其运行可能在 3 种情况下出

现问题。第一个情况是在产生数组的时候。一般而言没有任何方法可以为数组中的对象指定 constructor 自变量，所以几乎不可能产生一个由 EquipmentPiece objects 构成的数组：

```
EquipmentPiece bestPieces[10]; // 错误！无法调用 EquipmentPiece ctors。
EquipmentPiece *bestPieces =
  new EquipmentPiece[10];     // 错误！另有一些问题。
```

有 3 个方法可以侧面解决这个束缚。第一个方法是使用 non-heap 数组，于是便能够在定义数组时提供必要的自变量：

```
int ID1, ID2, ID3, ..., ID10;     // 变量，用来放置仪器识别代码。
...
EquipmentPiece bestPieces[] = {   // 很好，ctor 获得了必要的自变量。
  EquipmentPiece(ID1),
  EquipmentPiece(ID2),
  EquipmentPiece(ID3),
  ...,
  EquipmentPiece(ID10)
};
```

不幸的是此法无法延伸至 heap 数组。

更一般化的做法是使用“指针数组”而非“对象数组”：

```
typedef EquipmentPiece* PEP;     // PEP 是个指向 EquipmentPiece 的指针。

PEP bestPieces[10];              // 很好，不需要调用 ctor。
PEP *bestPieces = new PEP[10]; // 也很好。
```

数组中的各指针可用来指向一个个不同的 EquipmentPiece object：

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

此法有两个缺点。第一，你必须记得将此数组所指的所有对象删除。如果你忘了，就会出现 resource leak（资源泄漏）问题；第二，你需要的内存总量比较大，因为你需要一些空间用来放置指针，还需要一些空间用来放置 EquipmentPiece objects。

“过度使用内存”这个问题可以避免，方法是先为此数组分配 raw memory，然后使用“placement new”（见条款 8）在这块内存上构造 EquipmentPiece objects。

非常抱歉，打断了您的学习，也很抱歉上传的这本书并不完整，因为现在不劳而获的人太多，希望您能理解，在众多文件中您很幸运的下载了这本非常清晰的图书，既然您已经把这本书读到这个位置了，那么诚挚的邀请您加入 QQ 群：187541493，群内免费分享编程视频，高清 PDF 电子书，群内所分享的电子书均为重新整理过，目录详细，字体清晰，绝无歪斜。除部分经典书籍，收集整理的都是近几年的热销书籍。绝无陈旧书籍，滥竽充数。您也可以访问:www.wzbook.org 来获取完整免费的编程图书。祝您学业有成！