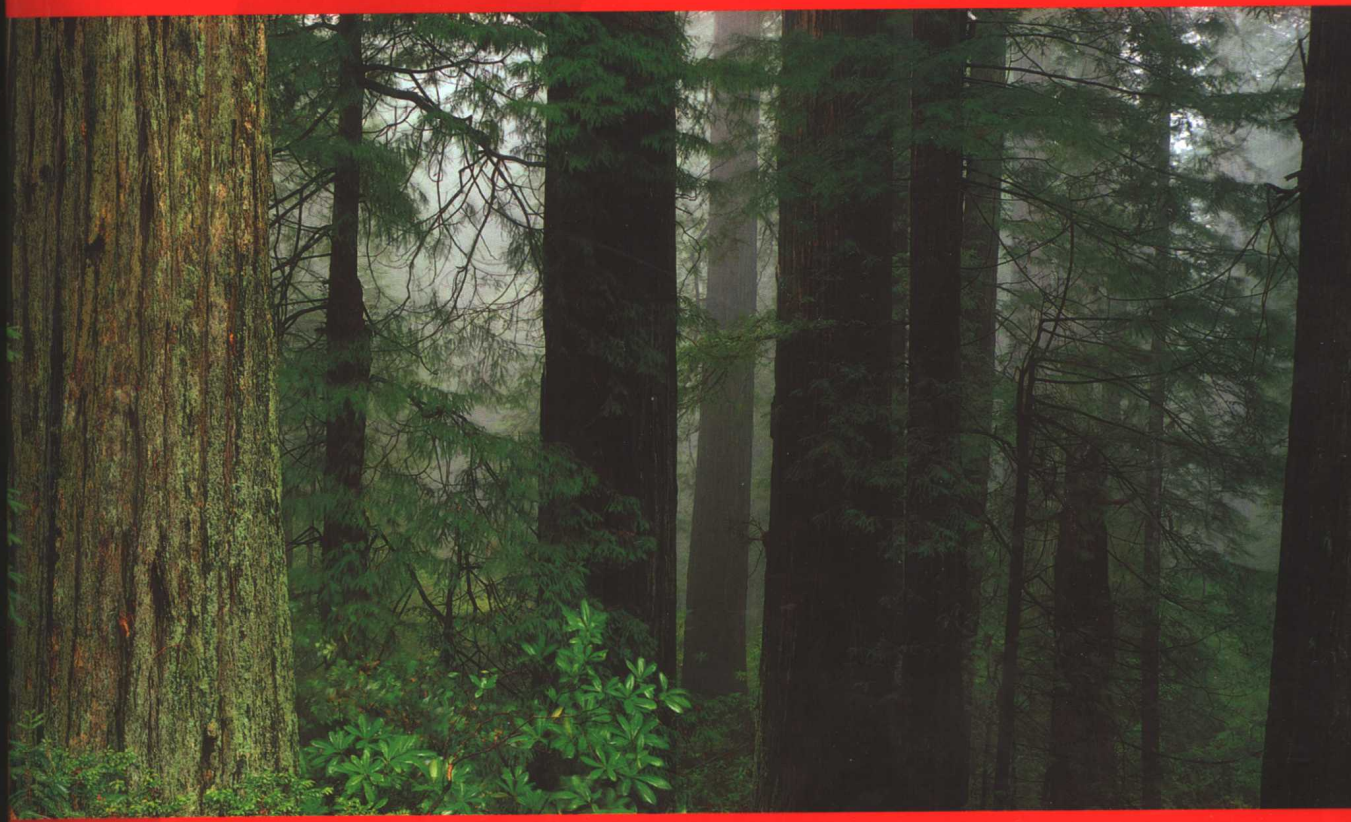
深入 C++ 系列

Advanced C++ Programming Styles and Idioms

Advanced C++ 中文版

[美] James O. Coplien 著
宛延闿 李石乔 苏文 等译
鲁玛勒 定 海 校



Addison Wesley

中国电力出版社
www.infopower.com.cn

深入C++系列

Advanced C++ Programming Styles and Idioms

Advanced C++ 中文版

[美] James O. Coplien 著
宛延闿 李石乔 苏文 等译
鲁玛勒 定 海 校

中国电力出版社

Advanced C++ Programming Style and Idioms (ISBN 0-201-54855-0)
James O. Coplien

图书在版编目（CIP）数据
Advanced C++中文版 /（美）考帕里安著；宛延闿 等译. 北京：中国电力出版社，2003
（深入 C++系列）
书名原文：Advanced C++ Programming Styles and Idioms
ISBN 7-5083-1915-X
Ⅰ.A... Ⅱ.①考...②宛... Ⅲ.C 语言－程序设计 Ⅳ.TP312
中国版本图书馆 CIP 数据核字（2003）第 117456 号

丛 书 名：深入 C++系列
书　　名：Advanced C++中文版
编　　著：（美）James O. Coplien
翻　　译：宛延闿　李石乔　苏文　等
审　　校：鲁玛勒　定海
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）88518169
印　　刷：北京市地矿印刷厂
开　　本：787×1092　1/16　　印　　张：20.75　　字　　数：531千字
书　　号：ISBN 7-5083-1915-X
版　　次：2004 年 2 月北京第 1 版　　2004 年 2 月第 1 次印刷
定　　价：36.00 元

译 者 序

当今 C++语言的使用已相当普及，它已深入到信息产业的各个领域。基于这种状况，对 C++语言的风格和习惯用法的研究，促进 C++语言的应用和发展是很有必要的。

C++是一种混合性语言，它既具有独特的面向对象特征，又保留了传统的高效结构 C 语言的主要特征。C++既能提供给程序开发者以面向对象思维开发的能力，又不会失去内存运行效率，并能在普通的计算机硬件上产生高质量的软件产品。但，C++更富于表现力蕴涵在 C++软件结构的语言风格和习惯用法上。程序设计语言的风格来自于语言专家的经验，这些经验是在实践中锤炼过的，从而形成了相对稳定的习惯用法。

学习 C++语言，像练习打球一样，首先基本功要扎实，然后才能在此基础上升华一步。我们翻译本书的目的，就是试图把他们多年来在语言实践中获得的 C++语言风格和习惯用法介绍给读者，以便提高读者利用 C++语言高效地解决实际软件问题的能力。

我们在翻译过程中进行了认真审核校对，订正了若干不严谨的地方，对一些常用的关键字词作了较确切的解释。对每章的练习题中不合适或不确切的地方作了删节，也增加了一些有用的练习题。其目的是为了更好地加强读者对这本书的概念和习惯用法的理解。

参加本书翻译工作的还有宛霞、甄炜、李保林、米慧敏、崔柯、石良秀、乔立琴、钟义、韩文智和蔡凤奇。王静、彭芬芬和张凤兰对本书中的程序做了不少测试、整理和校对工作。

希望本书对 C++语言的应用与开发能起到一定的推动作用。

宛延闿

2003 年 12 月于北京

序　言

本书是为有一定 C++开发经验的人员设计和编写的。为了更加精通 C++程序设计，我们不仅要深刻地了解如何学习一种新的程序设计语言，而且更重要的是如何利用这种语言高效地解决实际的软件问题。

学习程序设计语言

通常，在语言手册中，并非读者所要了解的东西都能完全描述出来。其实做任何事情都不可能面面俱到，学习程序设计语言也是这样。学习语言的语法可以使我们朝着更深层次的思维方向理解，但这仅仅是深化理解的开始，更重要的是大多数我们系统所要建立的程序结构的准则，应当表述设计概念的风格和习惯用法。

风格应当把优点和技能区分开来。一种高效的主体风格或者是高效的程序设计风格来自于个人的经验和在其他经验基础之上建立起来的风格。软件工程师应当知道如何选择与应用相匹配的程序设计语言，以编写出优秀的、结构良好的程序。为了达到这个层次的水平，我们需要超越一些规则，抛弃死记硬背的学习方式，最后达到概念和结构上的抽象，这就是本书所说的“高级”的含义所在。

程序设计的准则、约定和概念推动了所建立的系统结构，使我们更加清晰地了解到系统的模型是如何建立的。问题分解和系统组合的模型是一种范型，是把现实世界分为可管理部分的模式。C++语言是多重范型语言，C 程序员使用 C++就好像使用最好的 C 一样，面向对象主张事物的多态性。事实上，不同的方法通常必须表达对软件问题高效、优美的问题解。

学习程序设计语言就像学习一种自然语言一样，基本的语法知识可以让程序员在编写过程时更加简单，程序编写得也更加简洁，好像一个人仅从几千个词语的字典中就可以撰写出美丽动人的童话故事一样。

但学习语法和基本语义是需要花费相当多的时间的，不可能一下子就挖出个金娃娃。语言的奥妙属于另一类复杂的问题。学习程序设计语言与学习自然语言的差别就在于要学习语言的习惯用法。例如，在 C 本身不存在把构造

```
while(* cp1 ++ = *cp2 ++);
```

作为基本构造块来处理，而一个不熟悉这种构造的程序员却不曾发觉。

在程序设计中，像自然语言一样，重要的习惯用法，或者说好的习惯用法可以大大减轻程序员的工作量，就好像在任何一种语言中习惯用法丰富了相互之间的交流一样。程序设计的习惯用法是可重用程序设计的“表述”。在同样的意义下，类是设计和代码的可重用单位。简单的习惯用法（像上面 while 循环）是约定的表示，但很少是程序设计的核心。本书集中在习惯用法上，它影响到如何在整体结构中使用 C++语言并实现一种设计。这种习惯用法大大改善了对问题的洞察力，导致更

加简单的习惯用法的出现。通常，习惯用法引入了某种复杂性，但只要编写一次就可以随处运行了。一旦构造成功，程序设计语言习惯用法便蕴含在约定和技巧之中。

本书的方法

假定读者有C++基本语法的背景，本书试图把精通C++的专家们所获得的语言的风格和习惯用法传授给读者。本书展示了如何把 C++语言的不同风格用于简单的数据抽象、“羽毛丰满”的抽象数据类型的实现以及面向对象程序设计。与此同时，本书也探索新的、C++语言间接支持的核心习惯用法，诸如基本功能和框架的程序设计以及高级垃圾收集技术。

本书的结构

本书用围绕语言的特征来学习C++高级特征。从支持C++特征的观点出发，来考虑C++特征中不断增长的强有力的抽象。本书每个章节都围绕习惯用法的家庭来组织，习惯用法渐进地建立在相互之间有关联的后继各章中。

第1章提供了C++习惯用法的历史背景。该章提供了一些产生习惯用法的原动力以及能够作为C++语言一部分或作为语言外部使用的不同的习惯用法。

第2章介绍了基本的C++语言构造块：类和成员函数。虽然很多素材是基本的，但本章建立的习惯用法和公用程序在后面章节中会再次出现。从设计观点出发，本章还介绍了编译类型系统和用户定义类型与类之间的关系，同时也介绍了const的更有用的习惯用法。

第3章介绍了使用类的“完全”类型的习惯用法。C++已经自动演变成许多拷贝工作和初始化对象工作，但对大多数类来说，程序员仍然需要定制赋值和默认的构造函数。本章提供了定制的框架，描述了调用的规范格式即定义了在对象工作机制下的原则和标准。除了大多数共同使用的规范格式以外，还将习惯用法引入到对新的和已有的类的引用计数之中。本书最初的习惯用法超出了直截了当地应用基本的C++语法的范畴，在引用计数、计数指针上的变化方面导致了从C++中把计算机的灵巧的指针、类似于对象的指针舍去。最后，本章还寻找如何把对象的建立从初始化中分离出来，这对熟悉基本 C++的程序员来说，似乎是个不太自然的习惯用法：C++与紧密地耦合出现的两种操作相关联，在方法设计的驱动中以及在相互依赖资源的系统中需要分离它们的出现。

第4章介绍继承。

第5章添加了介绍面向对象程序设计继承的多态性。许多新的C++程序员正在变成“继承热”，在各种场合中都使用它。继承的确是大多数面向对象范型所支持的重要特性，它对软件可重性应用特别有用。第5章还介绍了继承与多态性的区别，以帮助读者识别这两种概念从而避免出现混淆。

第6章从结构和设计的剖面介绍C++的构造方法、风格和习惯用法。考察类在应用层面上的含义以及在高层语法上的含义。正确评价应用和抽象设计之间的关系，正确评价实现的类和对象之间的关系，以促进系统的健壮性并易于演化。另一个关键的进展是超越了具体应用的拓宽设计，覆盖了整个领域的应用。对这个领域分析的指导原则是本章的重要部分。本章还包含了适合继承使用的经验作法以及对不熟悉C++语言的程序员的难点领域等许多经验作法。经历过面向对象设计考验的读者将懂得如何把设计的输出转化为C++代码，作为继承替代物的封装性是C++语言上下文中的研

究对象，可重用和多态性也是其研究对象。

第 7 章研究代码和设计的可重用性，介绍 4 种不同的代码机制，特别值得关注的是“继承热”的优缺点。本章介绍了使用模板的参数化类库的一些习惯用法，以便减少代码生成量。

本书剩下的部分超出了 C++本身的范畴，进入到高级程序设计的习惯用法。

第 8 章介绍样本实例，以取代 C++类的许多重要角色。这些例子作为特殊对象来介绍，以解决某些共同的开发问题，诸如“虚拟构造方法”问题，而这些例子也给支持类独立性和开发独立性的更强有力的设计技术奠定了基础。

第 9 章集中在符号语言风格上，打破了许多 C++程序设计所保持的基本概念，包括强类型和显式内存管理。本章的习惯用法的确超越了 C++开发的主流范畴，是在 Smalltalk 和 Lisp 中发现并保留下来的风格。

第 9 章还介绍了支持增量式运行时更新的习惯用法。这种习惯用法的实现必须依赖于目标平台的许多细节。该材料的要点是使读者精通增量式装入问题必须工作的技术层面，这个例子是典型的。作为技术的进展，第 9 章的目的不是把 C++改造成 Smalltalk，这是不可能的，也是不应当的。这些习惯用法缺少对编译时间类型的安全检查，通常比“本机 C++”代码效率要低一些。所以，这些习惯用法一定要提供一定的灵活性和自动内存管理的增量式手段。

第 10 章覆盖了动态多重继承。多重继承是 C++有争论的特征。这种动态变化的讨论应避免传播到其他章节，而静态多重继承在第 5 章作为值来描述。动态多重继承避免了类组合的组合爆炸问题。我们已经发现这种方法在许多真实生活问题中是有价值的，它包含在编辑器、CAD 系统和数据库管理系统之中。

第 11 章讨论高层对象的系统性问题。本章产生 C++类的大板块以上的抽象层次，包括大型或特大型的软件构造、组织和管理。本章还将讨论大量的重要的系统性问题，包括调度、例外处理、分布式处理，还将讨论模块化和可重用的指南。本章把第 6 章和第 7 章紧密地联系在一起，在这些讨论之中也包含了对库结构和维护方面的考虑。

在**附录 A** 中，把基本 C++概念与 C 语言进行比较，许多读者已经学习过这些基本的或者能够找到介绍这些基本概念的文本。在这里包含这些材料有两个理由。第一，作为一种参考，当你需要澄清某些糊涂概念和结构时，不需要去费劲地寻找这样的文本。第二，C 和 C++风格是一种设计观点，表示如何融合过程和面向对象代码。这对 C++程序员的基本 C 代码的工作非常重要。

本书的例子虽然是基于 C++ 3.0 标准版本的，但对于目前许多其他平台以及其他一些 C++环境也很适用，对于 C++高版本显而易见是适用的。书中的实例基于通用目的，读者可以通过本书的例子举一反三，稍加修改即可应用到具体项目之中，或作为设计的框架，或作为完整的主体，许多通用的类都可以在本书中找到。

感谢

本书的出版都归功于我的朋友。首先，原动力来自于贝尔实验室 Chris Carson 的启示，无疑他是这本书的倡导老师，我感谢他主动积极的支持。本书还与 Keith Wollman 的指导，我的阅历丰富

的编辑以及其负责的管理主任 Helen Wythe 有关。Lorraine Mingacci 与我，对第 6 章进行了深刻又有意义的争论，并且我还与她对本书其余部分进行了详细讨论。本书还要归功于治学作风严谨的核心小组的多次讨论，他们贡献了许多有益的概念，这个核心小组成员有：Tim Born，Margaret A. Ellis，Bill Hopkins，Andrew Koenig，Stan Lippman，Barbara Moo，Bonnie Prokopowicz，Larry Schutte 和 Bjarne Stroustnp。Alexis Layton，Jim Adcock 和 Mike Ackroyd 提出了许多好的建议，使本书更加精彩，我深深地感谢他们。还有许多其他的贡献要归功于 Miron Abramovici，Martin Carroll，Brian Kernighan，Andrew Klein，Doug McIlroy，Dennis Mancl，Warren Montgomery，Tom Mueller，Anil Pal，Peggy Quinn 和 Ben Rovegno。幸亏借助于 Mary Crabb，Jean Owen 和 Chris Scussel 在文本格式上出色的经验以及 Brett L. Schucher 和 Steve Vinoski 辛勤的工作，得以在早期就发现了一些打印的错误；正是由于他们的努力，改善了打印的质量，最终避免了照相排版文档的缺陷。在确定具体章节标题上真的要特别感谢 Judy Marxhansen。

本书的出版不能忘记 AT&T 经理们的支持，是他们给了我充足的时间和资源。谢谢 Panl Zislis 和 Ben Rovegno 最宝贵的支持，并特别感谢 Warren Montgomery，Jack Wang 和 Eric Sumner, Jr.从道义上思想上的支持。

我还要说明的是从上我课的许多学生那里也汲取了许多好的营养，并反映到本书之中。特别感谢在 1989 年 Naperville，Illinois 和 Columbus，Ohio 以及 AT&T 贝尔实验室中我所教过的 C++课程的学生们。

目 录

译者序
序 言
第1章 引言 1
1.1 C++语言演变的历史 1
1.2 处理复杂结构的习惯用法 2
1.3 对象 3
1.4 设计和语言 3
练习 4

第2章 数据抽象和抽象数据类型 5
2.1 类 6
2.2 对象转换 8
2.3 构造函数和析构函数 9
2.4 内联（inline）函数 13
2.5 静态数据成员的初始化 14
2.6 作用域和 const 15
2.7 全局对象、常数和静态类成员初始化次序 16
2.8 类对象成员函数的 const 执行 17
2.9 指向成员函数的指针 19
2.10 程序组织的习惯约定 22
练习 23

第3章 具体数据类型 24
3.1 传统规范类格式 24
3.2 作用域和访问控制 29
3.3 重载：重定义操作和函数的语义 31
3.4 类型转换 34
3.5 引用计数：使用“可调内存”变量 37
3.6 操作符 new 和 delete 48
3.7 把初始化从实例化中分离出来 52
练习 55

第4章 继承性 57
4.1 简单继承 58
4.2 作用域和访问控制 63
4.3 构造函数和析构函数 71
4.4 类指针转换 73
4.5 类型选择域 74
练习 76

第5章 面向对象程序设计 78
5.1 C++运行时类型支持：虚拟函数 79
5.2 虚拟析构函数 84
5.3 虚拟函数和作用域 85
5.4 纯虚拟函数和抽象基类 87
5.5 信封和信件类 88
5.6 功能元件：作为对象的函数 110
5.7 多重继承 120
5.8 继承的规范格式 127
5.9 例子 130
练习 134

第6章 面向对象设计 135
6.1 类型和类 135
6.2 面向对象设计的实践活动 138

6.3 面向对象分析和领域分析 139
6.4 对象和类的关系 141
6.5 子类型、继承和转发 147
6.6 子类型、继承和独立性的经验规则 161
练习 162

第 7 章 重用和对象 163
7.1 所有模拟的分解放到何处 164
7.2 设计重用 165
7.3 4 种代码重用机制 166
7.4 参数化类型或模板 168
7.5 私有继承：继承支持重用吗 174
7.6 存储重用 176
7.7 接口重用：变体 177
7.8 重用、继承和转发 178
7.9 源代码重用结构的选择 179
7.10 在重用和对象上的概括 181
练习 182

第 8 章 C++中的样本程序设计 183
8.1 雇员样本的例子 185
8.2 样本和类属构造函数：样本共用的习惯用法 189
8.3 自主的类属构造函数 190
8.4 抽象基样本 191
8.5 关于样本习惯用法框架 193
8.6 相关的表示 195
8.7 样本和程序的管理 196
练习 197

第 9 章 在 C++中模拟符号语言的风格 202
9.1 增量式 C++开发 203
9.2 符号规范格式 204
9.3 通用收集类的例子 212
9.4 支持增量装入的代码和习惯用法 216
9.5 垃圾收集 223
9.6 基本类型的封装 230
9.7 在符号习惯用法下的多重方法 230
练习 233

第 10 章 动态多重继承 234
10.1 多重技术窗口系统的例子 235
10.2 防止误解的说明 237
练习 238

第 11 章 系统性问题 239
11.1 静态系统设计 239
11.2 动态系统设计 245
练习 254

附录 A C++环境中的 C 255
A.1 函数调用 255
A.2 函数参数表 256
A.3 函数原型 256
A.4 调用引用参数 257
A.5 参数个数不定 259
A.6 函数指针 261
A.7 const 类型说明符 262
A.8 C 代码的接口 264
A.9 操作符引用返回值 271
练习 273

附录 B Shapes 程序的 C++代码 274

附录 C 符号的 Shapes 284

附录 D C++中块结构程序设计 311
D.1 什么是块结构程序设计 311
D.2 构造基本块结构的 C++程序设计 312
D.3 有深度嵌套作用域块的选择 314
D.4 块结构的视频游戏代码 317

第1章 引 言

习惯用法是自然语言和程序设计语言的重要部分。本书的重点在于C++程序更富于表现力的习惯用法上和C++软件结构的语言风格上。诚然，在C++代码中也可以不需要这些习惯用法，但是风格和习惯用法可以更好地表述所编写的程序效率更高，或许还有美学的价值。这些都是多年来的语言实践经验和反复交流的结晶。

独特的习惯用法往往是很有趣的，但不是指令性的。这里，我们从语言历史的角度介绍两种类型的C++习惯用法。一种是影响C++发展的习惯用法；另一种是超越基本C++模式所支持的对象结构的习惯用法，风格和习惯都是很重要的，在面向对象原则下，习惯要服从风格。

1.1 C++语言演变的历史

我们知道，C++是一种混合性语言，它既具有独特的面向对象特征，又保留了传统的高效结构C 语言的主要特征。C++既能提供给程序开发者以面向对象的能力，而又不失去内存运行效率，并能在普通计算机硬件上产生高质量的软件产品。C++的祖先（带类的C）诞生于1980年，与当时开发的Smalltalk一样，早在1980年之前就有了“对象热”，之后便不断地成长。到了1983年夏天，“带类的C”已经进入到科研领域，并且继续起草和完成语言设计的最后定形方面的工作。

C++从用户实践经验中，从面向对象程序设计的大家族中，获得了有益的启示，使之在灵活性和可扩充性方面有了质的飞跃。程序设计在多重继承、封装技术和其他重要领域中有了新的突破。仅仅在1989年才开始标准化的C++语言在快速增长的面向对象语言的经验库上建立了适应性机制。

protected类作用域的建立是一个特殊的例子，它告诉用户如何开发C++并取得有影响的经验。friend关系被频繁地用于容忍导出类（derived class）违背基类的封装原则，在引入protected语言机制后，对friend的使用变得十分平常。这种公共反馈策略是在最好的时刻改善了C++保护模式，类似的经验为C++的多重继承和模板铺平了道路，并且无数次改善了语言的其他细节。

C++是以C为基础的，作为带类的C的改进，C++提供了编译时的类型一致性安全检查，也就是说，在面向对象语言中，C++既是高效的、又是安全的。但在C++演变过程中受到了成熟的对象范型和流行的C语言的影响，这两种影响不是（现在仍然不是）C++的目的。我们所确定的设计决策是在传统的C和采纳Smalltalk及Flavors新方向之间进行权衡，大多数决策是采用了最自然和最熟悉的技术。C++是Smalltalk年代的文化产物，并且其紧跟Simula（1967），Algol 68（1968）和C（1973）的语言技术。

在这些决策中的一个重要考虑是在哪里掩饰了高级特征的复杂性。对用户开发来说，力求使编

译中的复杂性尽量简化，使编译能够自动地与内存管理、初始化、清除、类型转换和输入输出（I/O）以及其他相关的日常工作有关。导致C++发展的另一个原则是编译不应该单纯地以损害用户方便使用为前提，但因重载的编译具有太多的智能成分而陷入了困境，即它既限制了用户对程序设计某些方面的单一模式（称为“武断”语言），又希望语言本身必须成长为表述交错功能的语言（称为“混乱”语言）。

1.2 处理复杂结构的习惯用法

我们希望提供一种强大的高层语言，以避免“武断”语言缺乏灵活性的缺点。为此，C++开发了组装块，在这个组装块中，程序员能够在他们自己的计算模块中编制代码。过去，语言特征大都依赖于编译时间的支持（即类型转换），而恰恰遗忘了语言本身。大量的内存管理、I/O模块和一些对象成员初始化等这些“语言特征”工作，在高级应用中是由用户自己来完成的。许多公共的程序设计工作变成了符合语言习惯的工作。即，使用C++构造函数规范在语言外部进行功能性表达，并列出语言所产生的一部分错误。

对象拷贝和内存管理的习惯用法是一个很好的例子。当一个对象被赋给另一个或作为函数参数来传递时，编译自动地产生指定的代码，或者从这些源对象中一个一个地初始化可接受的对象域。这种办法对大型对象来说是无效的，而且会导致指针数据成员的异常：当有时希望深度拷贝时，编译缺省时却提供浅度拷贝。这两个问题可以用引用计数对象的习惯用法来解决——把类划分为内存管理部分（称为处理）和应用部分（称为主体）。对对象的分配、初始化和删除，处理类由程序员提供代码版本，编译需要自动地应用管理内存的代码。在第3章我们将描述上面所说的习惯用法。

另一个例子是输入和输出。对于C来说，输入/输出不是语言的一部分。但这是用公共的左移和右移的重载操作支持输入和输出的流对象的习惯用法。所有类遵循纯的I/O约定。这里，复杂性不是在编译内，也不是在用户的应用程序中，而是在通用目的库中。该库是通过 #include 文件进行存取的，其中重载操作在库中出现，尽管逻辑移位操作在C++中通常是为I/O设计的。

这些习惯用法成功的关键在于对终端用户具有透明性，就好像这些习惯用法是语言本身的一部分一样。它们提供了“武断”语言的透明性和“混乱”语言的强大功能。用户使用内存管理模块一般不受什么限制，但要充分考虑到花费，用户需要选择一系列策略，比如要权衡效率、C接口的兼容性和在程序设计控制下的用户使用方便等。C++不会定义一系列策略，来困难地完成语言中更高级的集成内存管理模块。C++在作为元特征（meta-features）方面不是很丰富的，但组建成单一组装块的结构总是比仅仅是程序片的总的结构要强大。

这些元特征不是单一结构的产物，而是首先从研究机构获得实践经验，然后在开发项目中进行不断考察验证的结果。C++是具有“套件”机制的语言，它定义了强有力的公共程序设计工作的语义功能，语言的个别机制（诸如构造函数和重载操作）在它们自己的权限内具有某些价值，但这只是低层次的构造。把构造函数和重载分配操作组合起来的习惯用法提供了强大的高层语言。像语言的特征一样，许多习惯用法大多产生于早期C++版本的用户。根据面向对象应用中C++语言的演变以及新的C++特征在应用上有利的演变，在C++颁布的参考手册中所描述语言的定义达到了顶峰。这种规格说明自然充分地成为正式标准化的基础，而这些习惯用法已经考虑作为传统C++程序设计中公共实践的结果。

C++的用户或者作为“最好的C”学习的用户，对学习习惯用法不会感到困惑，程序设计语言也不会强迫他们使用习惯用法，习惯用法仅是尽可能提供组装块。从这种意义上说，习惯用法是“高级的”。本书的第一个目的就是覆盖C++传统的习惯用法，作为一种程序设计工具来使用。在我们的习惯用法的组织中，必须引入支持语言的特征，以便替代其他方式。其目的在于帮助C++程序员更快地过渡到符合语言习惯的高级构造函数，并提供一个合理的框架，以便使熟悉习惯用法语法的程序员能够过渡到更强有力的语义上来。

1.3 对象

以上所描述的传统的习惯用法的源头是 C++语言继承了 C，类对象的习惯用法不断地增长了C++用户的经验，自然会影响到其他程序设计语言家族。当今，C++仍旧是与 C 兼容的语言，它所具备的强大的面向对象特征，丝毫不会影响到其与 C 的兼容性。这是一个重要的征兆，即可以把面向对象构造用于传统 C 开发的编译、操作系统和实时控制系统之中。C++在支持面向对象程序设计语言上是充分的，虽然我们已经看到这种支持是通过约定、风格和习惯用法来实现的，但这些习惯用法的大多数仍是用语言元特征来支持的。

“面向对象程序设计语言”为软件产业的进步带来了生机，孕育出不断增长的需求。随着 C++的成长，用户会发现这种强大的面向对象的构造紧紧地跟上了面向对象技术的新的应用步伐。“虚拟构造函数”的需要就是一个例证。虚拟构造函数要求比 C++强类型模式更容易用弱类型语言来支持多态性。弱类型，或者动态类型是用间接和符合语言习惯的构造函数的附加层次来模拟的。这些习惯用法的大部分复杂性可以用在类中封装而加以解决。

对象是面向对象程序设计中的基本细胞，而习惯用法则是由这些基本细胞组成的类，并由类所支持的高级特征构成。习惯用法与开发 C++本身相比，在符合语言习惯方面稍微欠缺一些，即习惯用法在支持元特征方面不太多。还有一种说法认为，C++是靠继承 C 而来的，因而它不太能够完成当时的 C 和 Smalltalk 设计的哲理。但 Smalltalk 或 Flavors 在直接支持类型模型上要求在哲理上进行变更，而这些变更却偏离了对象模型的 C++模式，使之在“武断”语言和“混乱”语言之间陷入两难境地。实际上，我们已经看到，C++语言在面向对象领域上的应用远远地把 Smalltalk 甩在了后头。本书将表述的这些习惯用法及其经验，已经有效地应用在许多领域中，而无须专门语言的支持，并用在私有类成员中的封装实现其细节。

本书另外还要介绍 C++支持的许多必要的习惯用法，以及用 C++和 C++的传统习惯用法支持完整的对象模型。这些习惯用法不仅在 C++中出现，而且希望尽力仿效 Smalltalk 的优点，在任何强大的面向对象的 C++程序中，能够熟练而有效地解决设计和工程上的问题。

1.4 设计和语言

与语言演变一样，计算机科学已经正确评价了面向对象设计和面向对象程序设计语言之间的关系。很早以前，人们就使用对象范型作为程序设计的工具，提供抽象、封装和超越面向过程设计的灵活性。近年来，对象（更精确地说，是实体，它们的应用领域极为相似）已经成为设计构造的基本细胞。

C++遵循这种演变，C++的类概念是把初始设计映射到设计概念，通过私有继承追加可重用构造。像经验库的形成一样，C++子类型模式用面向对象分析和设计表达了对象和生成的类之间的关系，编译时强类型C++模式帮助我们避免滥用继承。

设计不是本书的重点，虽然设计上的考虑贯穿在本书的文字之中。设计和程序设计在对象范型中如此相似，即便本书是有关“程序设计”内容的，它也不可能忽略一系列设计方法上的考虑。第6章的重点是在C++语言的视图上；第7章讨论大量的设计问题；第11章讨论系统性问题，它超越了“确定对象”和正确评价继承的使用。像语言的语法、语义和习惯用法一样，这些设计方法都是多年经验的结晶。

C++发现它本身处于流行的面向对象设计之中，正如上面所讨论的，要获得成功必须付出很多的努力。大多数C++的习惯用法在下面几章加以介绍，它们都与对象范型有关。例如，第5.6节的一部分、第11章的子系统模块化技术和第7章的参数化可重用构造，这些都是C++的高级使用，也是语言中的精髓。

我们希望本书中的C++语言的描述、习惯用法和风格能给读者呈现出高级程序设计的一个侧面，并给读者一些有益的经验。不论是当前的工作还是未来的工作，风格和习惯用法对C++项目应用来说，都是相辅相成的。正如C++语言的著名计算专家所说，一个好的程序设计和好的设计是判断力、透彻理解和经验的结果。本书对大型系统如何使用C++语言开发具有较高的参考价值。

练 习

1.1 从C++演变历史中，你是如何看待站在“巨人的肩膀上”开发新的软件的？为什么Smalltalk-80这样纯面向对象语言没能够发展起来？为什么C++这样混合型面向对象语言反而蓬勃地发展起来了，这说明了什么？

1.2 C++和C之间是一种什么样的关系？

1.3 C++习惯用法对软件开发起到什么样的作用？

1.4 C++是什么样类型的面向对象程序设计语言？它的主要面向对象特征是什么？

第2章

数据抽象和抽象数据类型

类型程序设计语言在1950年就出现了，它不是一种程序设计的“当前最好的实践”，也不能当做系统结构的工具，而仅仅作为源程序中产生高效目标代码的手段。之后，程序设计语言类型系统被一致认为在界面类型检测和类型安全方面做出了一些有价值的工作。到了1960年，类型已成长为程序构造块的一个过程段。这些进展产生了新的、更加抽象的程序设计语言表示和构造，这就是：

- 抽象数据类型（ADT），它定义了数据抽象的界面，无须指定其实现细节。抽象数据类型可以用不同的实现来适应不同的需求，或者用不同的实现来适应不同的环境。
- 类（class）是抽象数据类型的实现。首先支持类的语言是Simula 67。
- 高级程序设计语言类型系统。ADT首先确立了语言的抽象地位，操作符重载和其他程序设计语言的进步赋予ADT一种表现力的新的层次和能力。这样的机制，首先出现在Bliss，Algol 68和Simula语言中，而在Mesa和Clu中更进一步进行了改进。

在C++中，内置类型（int，short，long，float，double，long double，char和指针）能在编译时进行界面检测并生成有效代码。这些抽象已预先封装在各个C++环境中，准备为计数和索引服务，同时也为数值分析、符号处理等构造块服务。C++类允许把相关的数据，它们的操作以及紧密耦合的功能与创建一个用语言支持的程序单位组合在一起。这些类有相同的能力，并且像内置类型那样方便使用。即它们可以用于说明和动态地创建新的实例，而它们的使用权取决于类型检测。类的界面可以分别管理它们各自的实现，类界面充当ADT规格说明的作用。

但是，正是因为某事或某物是一个类，这并不意味着可以用同样的概念来处理像int，double一样的内置类型。类的设计者必须提供与新类相吻合的构造来纳入编译类型系统，而编译能够告知如何在内置double和用户定义的Complex（复数）类的实例之间进行转换，编译也能告知用户如何产生建立、赋值和拷贝类的实例的有效代码。更进一步地，一个类可以转换成用户定义的类型，我们称之为“具体数据类型”（concrete data type）。这样的类对象就具有说明、赋值和准确地传递参数的能力，就如同C和C++的内置类型的变量一样。

类是用于设计和实现的抽象，甚至无须构造具体的数据类型。把一个类转化到具体数据类型的工作不太多，而使用具体数据类型却十分方便，通常把一个类转化成具体数据类型的努力是值得的。但简单类是我们研究的起点，本章将首先讨论基本类，读者将用自己的方式构造重要类的类型。第3章介绍把本章精心构造的基本类类型转换成具体数据类型的过程。

本章从介绍类的架构开始，这是C++最基本的抽象机制。我们将看到如何把功能和数据组合起来构造一个类，即彻底地按照C的方式来做这件事。初始化的构造函数和类变量的析构函数我们将在下面加以描述，本章也介绍其他C++构造和概念，这些与内联（inline）函数、const成员函数和

const 对象、静态成员函数和成员函数指针有关。

2.1 类

C++类构造与 C 结构（struct）紧密相关。C++接受 C 语言说明的 struct 的风格和维护 C 语言的语义。通常，保留字 class 强调的是作为用户定义的数据类型的一个结构，而不是简单的数据聚集。也就是说，当数据和功能组合在一起时，就应使用 class；仅当数据撮合在一起时，需要使用 struct 来替代。把一个类中的功能划分成组，每个功能称为成员函数，每个成员函数是这个类所包含的“成员”。C++的 struct 就好像一个 class，可以有成员函数，但更多的是保留了 C 语言非智能数据记录的思想。在 C++中，class 与 struct 之间的不同仅在于缺省的访问权限。正如例 2.1 中所描述的，当用于数据封装机制时，C 结构和 C++类之间具有一定的相似性。

例 2.1：C 结构和 C++类之间的相似性。

C 代码：

```
struct {
    char name [NAME_ SIZE];
    char id [ID_ SIZE];
    short yearsExperience;
    int gender:1;
    unsigned char dependents:3;
    unsigned char exemptions:4;
} foo;
struct {
    char name [NAME_ SIZE];
    char id [ID_ SIZE];
    short yearsExperience;
    int gender:1;
    unsigned char dependents:3
    unsigned char exemptions:4;
} bar;
```

C++代码：

```
struct {
    char name [NAME_ SIZE];
    char id [ID_ SIZE];
    short yearsExperience;
    int gender:1;
    unsigned char dependents:3;
    unsigned char exemptions:4;
} foo;
class {
public:
    char name [NAME_ SIZE];
    char id [ID_ SIZE];
    short yearsExperience;
    int gender:1;
    unsigned char dependents:3;
    unsigned char exemptions:4;
} bar;
```

与 C 不一样，C++对每个带标记的结构或类引入新的类型到程序中，它是通过 typedef 插入到说明之后的：

C 代码：

```
struct EmpLabel {
    char name [NAME_ SIZE];
    char id [ID SIZE];
    int gender:1;
    ...
} foo;
typedef struct EmpLabel Employee;
struct EmpLabel fred;
Employee lisa;
```

C++代码：

```
struct Employee {
    char name [NAME_ SIZE];
    char id [ID_ SIZE];
    int gender:1;
    ...
} foo;

struct Employee fred;
Employee lisa;
```

在 C++中，标记和类型是同义词，而在 C 中它们是相互独立的名字。

C++结构的行为像类一样，其成员缺省时输出是公有的，而类成员缺省时是私有的。用保留字 private，public 和 protected 可以改变访问权限。当这 3 种保留字之一出现时，后面要紧跟着冒号，指示以下成员在此控制范围之内，直到该类结束或到下一个这样的指示出现为止。缺省控制的有效范围是在下一个这样的指示之前，控制属性有以下语义：

- 对象的 public（公有）成员能访问对象类说明的任何功能，其访问作用域是对象本身。
- 对象的 protected（保护）成员仅能访问对象类或其导出类的成员函数。导出类对象不能访问基对象类的 protected 域，甚至不是它自己基类的一个对象。导出类成员函数仅能访问 protected 成员，这个成员仅是它自己类对象的基类的一部分。
- 对象的 private（私有）对象仅能访问对象类的成员函数。

如果用户愿意的话，可以使用违背（由 private 和 protected 提供）封装原则的 friend 机制。这部分内容将在后面详细加以讨论。

虽然可以把类（class）作为结构（struct）在语法上扩充，但我们还是希望根据程序设计的通信思想把类更进一步地加以扩充。在任何程序结构技术中，结构习惯用于与数据相关的包。例如，在功能分解中（见附录 D），功能的每层有它自己的数据，而 struct 可以用于在给定层次上把相关数据组合在一起（或者建立几个相关数据分组）。在功能分解中产生的数据结构不一定映射到应用程序的抽象状态中。

C++类超越了 C 的结构，提供了两件东西，即类型和抽象。在 C++程序中，习惯用类创建新的用户定义类型，这个类型我们称之为调用抽象数据类型。类是类型的组装块，因为它们获取了具有一定行为的数据集合的活动。例如，考虑下面一个 Complex（复数）成员类：

```
class Complex  {
public:
friend Complex& operator+ (double, const Complex&);
       Complex& operator+ (const Complex&) const;
friend Complex& operator- (double, const Complex&);
       Complex& operator- (const Complex&) const;
friend Complex& operator* (double, const Complex&);
       Complex& operator* (const Complex&) const;
friend Complex& operator/ (double, const Complex&);
       Complex& operator/ (const Complex&) const;
       double rpart() const;
       double ipart() const;
       Complex(double = 0.0, double = 0.0);
private:
       double realPart, imaginaryPart;
};
```

这里，类提供了不仅能获取 realPart 和 imaginaryPart 之间关系的一种方式（当然，作为一个 struct 也能做到），而且能获取一个 Complex 的成员表示与其行为之间的关系。C 语言的构造不可能方便地获取这些关系，而且也不安全，但却能够由程序设计来获取它们的表示。

类帮助程序设计者进入更高层次的程序设计构造，这些构造是为抽象服务的，而且这种抽象是与应用紧密相关的。类的使用重点是从应用领域抽象映射到问题领域的抽象，而在数据的 struct 方式中是办不到的。