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.c

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

Advanced C++ Programming Style and Idioms (ISBN 0-201-54855-0)
James O. Coplien
Authorized translation from the English language edition, entitled Advanced C++ Programming Style and Idioms, published by Addison Wesley, Copyright©1992
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 the Publisher.
CHINESE SIMPLIFIED language edition published by China Electric Power Press Copyright©2003

本书由美国培生集团授权出版。
北京市版权局著作权合同登记号 图字：01-2002-4851 号

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
版权所有　翻印必究

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。

一个优秀的程序设计工作者可以根据抽象数据类型进行思考，并根据结构和功能来编写代码。在这种情况下，一个好的C++程序结构与一个好的C程序结构是相同的，这就是说，要“从里到外”进行转换。我们称此为“对象转换”，下一节我们将讨论“对象转换”。

2.2 对象转换

许多程序都有库或围绕一个数据结构组织的模块的支持。例如，构造一个简单的栈（Stack）。该栈的数据可以放在一个 struct 之中，并且可以有几个过程与这个结构有关，当然可以操作在这个结构说明的变量上。

在C++中，我们要彻底地改变这种状况。我们所看到的功能是属于结构的，因为是将它们的数据和结构紧紧耦合在一起，用与之相关联的过程传递一个 Stack 结构来替代，将这个过程与其数据连在一起放在 struct 内。这些功能被称为成员函数或操作，其他语言（如 Smalltalk）称它们为方法（method）。例 2.2 表示了按此风格构造的栈的例子以及功能性代码。之所以说是简单栈，是因为这个栈的长度是固定的，类型也相同，都是 long 型的。

例 2.2：相应的 C 和 C++实现的栈构造。

C 代码：

```
#define STACK_ SIZE 10
struct Stack {
    long items [STACK_SIZE];
    int sp;
};
void Stack_ initialize(s)
struct Stack *s;
{
    s ->sp = - 1;
}

long Stack_ top(s)
struct Stack *s;
{
    return s->items [s ->sp];
}
long Stack_pop(s)
struct Stack *s;
{
    return s->items [s->sp--]
}
void Stack_push (s,i)
struct Stack *s; long i;
{
    s->items [+ +s ->sp] = i;
}
int main()
{
    struct Stack q;
```

C++代码：

```
const int STACK_ SIZE = 10;
class Stack {
private:
    long items [STACK_ SIZE];
    int sp;
public:
    void initialize ();
    long top () const;
    long pop();
    void push (long);
};
void Stack::initialize () {
    sp = -1;
}
long Stack::top () const {
    return items [sp];
}

long Stack::pop () {
    return items [sp--];
}

void Stack::push (long i) {
    items [++sp] = i;
}

int main ()
{
    Stack q;
```

```
    int i;
    Stack_ initialize (&q);
    Stack_push (&q,1);
    i = Stack_ top (&q);
    Stack_pop (&q);
}
```

```
    q. initialize ();
    q. push (1);
    int i = q. top();
    q. pop ();
}
```

注意，C++的成员函数，如栈的 push 函数（调用 Stack::push）没有间接引用结构指针，而且甚至不表示任何显式访问类对象，这对应于 C 实现中的 struct。这是因为 C++为类创建了作用域的另外层次，而且功能可以认为是放在该结构的实例内的。与 struct 一样，一个栈（Stack）“变量”的每个说明都需分配内存的某一空间。例如，在 C++程序的 main 中的实例 q 对应于在运行时栈上的内存。这个实例就称为类 Stack 的一个对象，Stack 的成员函数被用在它的对象上（如 q.initialize()或 q.push(1)）。在这些涉及类 Stack 的成员函数的表达式中，使用的是那个被调用的成员函数的对象数据（这里是 q）。

成员函数通过隐式参数指向被调用函数的对象来访问对象数据，参数名是 this。虽然 this 作为一个参数被清晰地传递，但程序员仍然能够直接访问它。this 参数执行了与在 C 函数中结构 Stack 指针参数相同的功能，这个指针通常是编译时自动提供的。此外，编译自动地用预先标识符 this->，在成员函数内对类成员引用。在 Stack::initialize 内，表达式 sp=1 的含义是 this->sp=1。显式间接引用是合法的，但一般不这么做，因为通常会产生冗余。下面两种 C++代码段是相互等价的：

```
void Stack::initialize() {
     sp= -1;
}
long Stack::top() const {
     return items[sp];
}
```

```
void Stack::initialize() {
     this ->sp = -1;
}
long Stack::top() const {
     return this->items [this->sp];
}
```

注意，在成员函数 Stack::top 上 const 的指示，这是一个表示成员函数不可以修改被调用的对象（用 this 指向的对象）。有关 const 的构造将在第 2.9 节中详细加以介绍。

2.3 构造函数和析构函数

C++语言设计的两个重要的思想是：

（1）变量应总是能被自动初始化；

（2）类应能控制自己成员的内存分配。

为了把这些思想集成到语言当中，允许类的创建者和谐地安排分配或取消内存存储区，类的用户可以不必担心。为此，我们定义了特殊的成员函数，使对象的初始化和对象删除可自动地实现。这两个成员函数分别称为构造函数和析构函数。

编译具有一种识别构造函数和析构函数之间的专门的关系，即是用它们不同的名字来识别。构造成员函数的名字与这个类有相同的名字，而析构函数的名字仅在该类名字的左上角加上“～”即可。

在“本机”模式中使用 C++时（无须任何高级习惯用法），不能直接调用构造函数和析构函数。当需要初始化时，编译自动地产生构造函数和析构函数以初始化变量；当需要放弃之时，就自动地

清除所占内存。例如，对于局部类对象变量函数的入口，构造函数的这些变量自动地被引用。构造函数和析构函数也与内存管理原语 new 和 delete 一起，动态地初始化和收回所分配的内存。

注意，构造函数和析构函数是无返回类型的成员函数。这是因为它们绝不真实地直接调用，而是由编译产生代码，需要时才调用之。一个较为诱人的看法是，构造函数“返回”一个完全的对象，而构造函数恰好请求为对象留出初始化已有的内存块（例如，全局对象），作为初始化一个动态分配的对象（例如，当使用操作符 new 时）。第 3 章将描述在特殊情况下的高级内存管理，大部分情况下还是自动地产生调用。

如果刚好使用简单数据聚集或抽象的话，那就不一定要类的构造函数和析构函数。如果要求有动态内部结构的类行为；像程序设计语言的自然类型一样，则需要构造函数和析构函数。例如，上述的 Stack 例子就没有构造函数和析构函数。但有 initialize（初始化）函数，而这个函数必须在合情合理的状态下显式地调用设定一些事情。我们可以用构造函数来替代这个函数，其目的是要 Stack 初始化自己的内部结构，用户不必对 Stack 担心什么。首先，我们必须把该类的说明作一些变更：

```
class Stack {
public:
    Stack();
    long top() const;
    long pop();
    void push(long);
private:
    long items [10];
    int sp;
};
```

然后，用构造函数来代替 initialize 函数：

```
Stack::Stack() {
     sp = -1;
}
```

这时，initialize 函数不再由 main 程序调用。当创建 Stack 对象时，若遇到 q 说明，则构造函数就会自动地被调用。这样，可以将调用 initialize 函数从 main 程序中去掉，此时程序会工作得更好。

例 2.3：具有构造函数和析构函数的一个栈（Stack）的说明。

```
const int STACK_SIZE = 10;
class Stack {
public:
    Stack();                // 一个构造函数
    Stack(int);             // 另一个构造函数
    ~Stack();               // 析构函数
    long top() const;
    long pop();
    void push(long);
private:
    long *items;
    int sp;
};
```

现在让我们来看更一般情况的栈类（例 2.3），进一步展示构造函数和析构函数。当前的 Stack 有 10 个固定长度的元素，我们可以创建一个 Stack，希望在说明时有比这 10 个少点儿或比这 10 个多点儿的元素。为了达到这个目的，我们就要定义一个带有栈尺寸参数的构造函数。这个构造函数可以动态地用 C++的 new 操作符来分配内存：

```
Stack::Stack(int size) {
    items = new long [size];
}
```

因为构造函数动态地分配内存，析构函数增加当对象清除时把内存收回到空闲池中。析构函数用 delete 操作符的动态内存指针来完成这些操作：

```
Stack:: ~Stack() {
    delete [] items;
}
```

这里，使用花括号把 delete[]items 的语义通知编译器，表示 items 指向 long 的向量，而不是单一地动态分配一个 long 的实例。

注意，现在有两个构造函数，它们具有相同的名字!名字 Stack 变成了具有两种含义的“已装载”，这就是重载，它不会引起困惑和二义性。如果上下文要求这个构造函数有一个自变量，那么就调用这个有一个自变量的函数。否则，调用另一个构造函数，而且我们能够定义缺省大小为 10 的栈。编译要维护这种自动化。在例 2.3 中已表示了新的 Stack 类的说明，而例 2.4 则展示了它的实现。

例 2.4：具有构造函数和析构函数的一个 Stack。

```
Stack::Stack(){
    items = new long[STACK_SIZE];
    Sp = -1;
}
Stack::Stack(int size)  {
    items = new long [size];                // 像类型 Sbrk 和 malloc 调用一样，
                                            // 如果存在，除非调用构造函数
   Sp = -1;
}
Stack:: ~Stack() {
    delete [] items;                        // 像空闲池一样，除非调用析构函数
}
long Stack::top()const {
    return items [sp];
}
long Stack::pop() {
    return items [sp--];
}
void Stack::push(long i) {
    items [--sp] = i;
}
int main()
{
    Stack q;                                // 调用 Stack::Stack()
    Stack r(15);                            // 调用 Stack::Stack(int)
```

```
    q.push(1);
    int i = q.top();
    q.pop();
}
```

如果一个类没有构造函数，则不能保证类的对象初始化。如果没有构造函数的类包含了这样一个有缺省构造函数的类对象的话，那么这些对象将合法地建立并且初始化。但没有构造函数的对象的其他域则未定义这些内容。例 2.5 就具有这样的代码。类 B 的对象没有作为成员的类对象，仅有内置 C 类型的实例（指向 char 和 short 的指针）。但是，B 的构造函数就会费力地初始化它的所有的对象成员。C 的实例有一个类对象成员 B 和一个 int；类 D 的对象仅有 int。主程序（main）创建了类 C 和类 D 的对象，我们注意到，类 C 和类 D 的对象所有基本的类型域都未初始化。如果这些类有构造函数，而且该构造函数指定这些域的初始化，那么初始化就完成了。

例 2.5：基本类型类成员的初始化。

```
class B {
public:
    B() {p = 0;s = 0;}
    int f();
private:
    char *p;
    short s;
};
class C {
public:
    int g();                // 没有构造函数
private:
    int i;
    B b;
};
class D {
public:
    int h();
private:
    int j,k;
};
C gc;                       // 初始化 p 和 s,i 置为 0
int main() {
    C c;                    // 初始化 p 和 s,i 待定
    D d;                    // 未初始化 j 和 k
    int 1 = c.g();
}
```

以上可以注意到 C 有一个 B 成员，因为 B 有一个构造函数，每当创建 C 对象时，就初始化 C::b 域。B::B 构造函数从缺省的构造函数（无参的）中被调用，编译为类 C 自动地产生这个缺省的构造函数。编译必须提供这个构造函数，所以要安排调用 C::b 的构造函数。但是，编译所产生的代码仅初始化已存在的那个构造函数的成员，所以仅有类成员在构造函数缺席的情况下才隐式地初始化。说明内置类型的成员不能被自动地初始化，这种行为是与 C 结构中缺少初始化是一致的。

2.4 内联（inline）函数

内联函数修饰符 inline 习惯上用于要求函数内联扩展。内联扩展避免了函数重载，这个函数是在调用点上用扩展该函数主体的方法来调用函数重载。编译将试图对使用前说明的函数代码进行内联扩展，其中说明和使用在同一源文件流中出现，即是相同源文件的一部分或用那个文件的#include源文件的闭包。这样，内联扩展能够极大地节省 CPU 时间（大约能节约四分之一时间）。

C 的预处理宏已经长时间习惯用于模拟 C 中的内联扩展函数。下面是宏习惯模拟绝对值函数：

```
#define abs(x) (x<0?(-x):x)

int main(){
   int i,j;
   ...
   i = abs(j);
   ...
}
```

内联函数可以替代这样的预处理宏的定义。内联比宏更可取，因为内联遵循函数同类型和作用域规则，内联产生的代码也比宏好。在某些环境中，可以提供比宏更好的符号调试功能。例如，某些 C++ 环境允许处理带有编译标记的内联函数扩展，其目的是将所有函数编制到子程序代码拷贝之中。这意味着调试程序能够很容易地在这些函数上设置断点，通常在宏中却是不能做到的。

一个函数可以显式地在函数定上使用 inline 保留字构造一个内联函数：

```
inline int abs (int i) {return i<0?-1:i;}
class Stack {
    ...
   long pop();
};
inline long Stack::pop() {
    return items [sp--];
}
```

成员函数可用定义说明的一部分来构造一个内联函数；这种方法我们不鼓励，因为它使管理趋于复杂化（见第 2.10 节）。但是，具有说明定义的表示在本书中将被使用，因为这种表示很清晰，无须任何内联扩展的意图。这里有一个内联扩展成员函数的例子：

```
class Stack {
   ...
   long pop() {return items [sp--];}        // 内联扩展
};
```

内联函数的定义（当然，要对这个函数进行内联说明）必须发生在源文本中，在任何调用该函数之前定义，否则是错误的。考虑下面的例子：

```
class C {
public:
   inline int a()    {return 1;}
   int b()           {return 2;}          // 也是内联
   int c();
```

```
    int d();
    int e();
};
inline int C::d() {        // 错误：C::c 之前被调用
        return c() +1;     // 以下是内联的说明
}
inline int C::c() {return 0;}
int C::e() {
        return c ()+5;     // C::c 调用内联扩展
}
int main() {
        C o;
        int i = o.d();     // c::d 内联扩展，
                           // 但嵌套 C::c 不调用
}
```

不同的 C++在内联扩展实现上有它们自己的限制。例如，编译不能扩展循环函数或递归函数。

内联函数不是改善性能的灵丹妙药。通常，改善性能必然增加代码量，扩展因素依赖于所使用的 C++环境。有一个 80/20 规则（也称“黄金”规则），即寻找程序中 20%的代码要花费 80%的时间，内联扩展函数从内部循环中调用。确定一个程序中有没有好的内联函数的标准是方便和改善性能的方式。

2.5 静态数据成员的初始化

C++类可以有静态数据成员，它们与静态成员函数的构造是平行的。静态数据成员在类的界面中加以说明，通常在头文件中（此文件具有.h 后缀）。这些数据的定义应在.c 文件，或.C，.CPP 文件中，依赖于系统使用的是否是 C++源程序文件，不在头文件中。例 2.6 给出了这样一个例子。

例 2.6：静态数据成员的初始化。

头文件的说明：

```
struct X {
    X (int,int);
    ...
};
struct s {
    staic const int a;
    staic X b;
    ...
    int f();
};
```

在.c 文件中定义

```
const int s::a = 7;
X s::b(1,2);
int s::f() {return a;}
```

C++静态成员函数与全局函数一样，其作用域在这个类中。它们与全局函数一样可以有效地被调用，但比通常成员函数的调用更有效一些：

```
class X {
public:
   void foo() {fooCounter ++;...}
   Static void printCounts() {
       printf("foo called % d times\n",fooCounter);
   }
private:
   Static int fooCounter;
};
int X::fooCounter = 0;
int main() {
   printCounts();          // 错误（除非真的存在这样的全局函数）
   X::printCounts();       // 正确
}
```

静态成员函数大部分典型使用是管理类的所有对象的公共资源，静态成员函数经常操做在静态类数据上。例如，一个类可以包含其成员函数被调用多少次的计数。每个成员函数可以增加它自己的 static int 在线计次的次数。静态函数习惯用于复位、检索或打印这样计数的值。

另一个静态成员函数潜在的使用可以把相关函数组合在一起，这些函数不应该当做对象来处理。这样的数据之间的关系可以比大多数对象成员函数组合在一起要弱，而且它们与设计层次资源无关（见第 6 章）。在某种意义上，这种模块是 Ada 包的概念，这样的构造我们将在第 11 章中加以介绍。

2.6 作用域和 const

在 C 中，#define 构造习惯用于带有符号名的常数值（称为声明常数）。参照这种思想，C++中使用 const 符号达到了同样的目的。C++常数可以有作用域，常数的作用域在这个类中。

常数用于一个局部文件中，在这个文件中说明，在这个文件中定义。在同一程序中，对其他文件是不可见的：

```
Const int MAX_STACK_SIZE = 1000;
Const char *const paradigm = "The Object Paradigm";
```

常数用 extern 说明可以在交叉源文件中共享，这些说明要放在头文件中公共说明：

```
extern const double e;
```

并且在程序源文件中（.c 或.C 或.CPP 文件），其值是放在单一定义中的：

```
#include <math.h>
extern const double e = exp(1.0);
```

const 静态成员的语法比在一个类内的符号常数说明要复杂。特别是，静态成员常数的值可以不与编译时间挂钩。这就意味着这样的值可以不说明数组的长度：

```
static const int SIZE1 = 10;
class C {
    ...
    static const int SIZE2;
```

```
    char vec1 [SIZE1];          // O.k.
    char vec2 [SIZE2];          // 非法,SIZE2 不知道
};
const int C::SIZE2 = 10;
```

为了克服 const 的这个缺点，我们使用一个替代 const 的构造，那就是枚举（enum）类型，enum 是不可改变的，这个符号的整数值用于 const int 不能履行的一些作用。在 C++中的枚举类型与 C 一样，通常是定义符号整常数的聚集，可以用 switch 语句和 if 语句实现。它等价于在 enum 说明中赋予这些符号以具体值，给出它们这样的符号常数一个具体的含义。这里，我们在类 Stack 内创建一个命名为 StackSize 的“常数”：

```
class Stack {
public:
    void push (int);
    int pop();
    int top()const;
private:
    enum {StackSize = 100};   // const 模拟
    int rep[StackSize],sp;
};
void Stack::push(int el){
    if (sp ->= StackSize)error("stack overflow");
    else rep[sp++] = el;
}
```

2.7 全局对象、常数和静态类成员初始化次序

C++确保变量初始化按照给定源文件内出现的次序进行，但不能保证全局类对象在交叉源文件中初始化的次序。这适用于所有全局类对象的变量，特别对静态类成员有一定的价值，因为全局类对象的初始化可以依赖于类的静态常数值的有效性。

有关初始化次序的假定可能会导致难以想像的缺陷。例如，考虑一个简单的头文件 Anglc.h，其说明如下：

```
#include <math.h>
#include <stdio.h>
extern const double  pi;
class Angle {
public:
    Angle (double degrees) {
        r = degrees *pi/180.0;
    }
    ...
    void  print () {
        printf("radians = % f\n",r);
    }
private:
    double r;
};
```

现在考虑源文件 pi1.c，它包含了常数 pi 的定义：

```
#include <Angle.h>
extern const double pi = 4.0*atan(1.0);
```

而源文件 pi2.c 包含以下应用代码：

```
#include <Angle.h>
Angle northeast = 45;
int main() {
    Angle northeast2 = 45;
    northeast.print();
    northeast2.print();
    return 0;
}
```

当在系统上编译和运行时，这个程序就产生以下输出：

```
radians = 0.000000
radians = 0.785398
```

因为 pi 在调用 northeast 构造函数的时候还没有被初始化。

如果存在合法的需要特意安排这样的初始化次序的话，那么第 3.7 节中所介绍的技术就可以被使用。最好尽可能避免这些依赖性，特别是在使用其他库代码的时候。

2.8 类对象成员函数的 const 执行

从 C++语言 2.0 版本开始，已经产生了某些 const 的新的应用。在早期的 C++版本中，它可以用调用成员函数的方法来改变 const 值，这种做法已经作为一个问题和不足被正式报告了多次。

如果不允许调用 const 对象的成员函数的话，那本身就执行了无用的 const 对象。基于语言的完整性，从 2.0 版本开始就引入了 const 成员函数，成员函数可以用 const 对象来调用。编译保证 const 成员函数不修改被调用的对象，也保证只有 const 成员函数才使用 const 对象。例 2.7 表示，用函数说明的参数表后缀来使用 const，将它设计成与*.后缀的 const 使用相一致。因为是分开编码，所以，当一成员函数作为 const 对象以修改被调用对象时，我们不能依赖于编译本身（不需要用户提供技巧）的检测。

例 2.7：const 成员函数。

```
struct s {
     int a;
     f(int aa) {return a = aa;}
     g(int aa) const {return aa;}
     //   h(int aa) const {return a = aa;}
};
void g()
{
     s o1;
     const s o2;
     o1.a=1;
     //   o2.a = 2;
```

```
        o1.f(3);
        //   o2.f(4);
        o1.g(5);
        o2.g(6);
    }
    /*
    删除 // 并且编译应该产生以下这些信息:
    Remove, line 5:error:assignment to member s::a of const struct s
    Remove, line 13:error:assignment to member s::a Of const s
    Remove, line 15:warning:non const member function s::f() called
                    for const object
    */
```

逻辑 const 和物理 const

对象的状态（即所有其成员数据的值）如何控制它的成员函数的动作呢？我们把对象状态分为3个部分：

（1）反映应用状态的数据（例如，记住瞬间到达的信息或信号）；

（2）松散耦合应用的数据，但其解是人工的（例如 Stack::sp）；

（3）支持调试和管理的数据。这些数据与应用的语义关系不大（例如，如何计算函数被调用的次数）。

显然，这不是最好的分类。这样的引用计数的数据可以通过第二目录或通过第三目录来计算，而其他状态可以杂乱地伸展到开头的两个目录中。但它可以帮助我们着重考虑成员函数与 const 对象之间是如何相互影响的问题。

再一个例子是，我们考虑一个类 BinaryTree（二叉树），它具有成员函数 findNode，根据某些规则返回一个结点的拷贝。当结点搜索次数过量时（用在树中结点数的临界值来鉴别），这个算法可以转换成**再平衡树**。我们期望 findNode 是 const 成员函数。如果它说明成这样，则它需要对再平衡树进行编译，即 const 成员函数不可以对它们的对象数据进行修改。

这种情况不会经常出现。这个技巧是通过 this 指针指向别名的方式来访问局部对象：

```
T BinaryTree::findNode(String key)const {
    ...
    if (needReBalance) {
        // 这里加上注释用以解释为什么违背了 const
        BinaryTree *This = (BinaryTree*)this;
        ...
        This ->left = This->left ->left;
        ...
    }
}
```

另一种方式是建立一个对 this 的引用，其行为是属于别名的，但具有较少的保护特性：

```
T BinaryTree::findNode (String key) const {
    ...
    if (needReBalance) {
        typedef BinaryTree *BinaryTreePointer;
        const BinaryTreePointer &This = (BinaryTree*)this;
```

```
            ...
            This ->left = This ->= left ->left;
            ...
        }
    }
```

第二种方式强调了 this 的常数别名的作用，所产生的代码可能与第一种情况相同。如果该函数是内联的，这两种方法之一便可以产生更少量的代码，这要依赖于编译的实现。

String 类列举了以上例子的对照法。String 有一个作为指针的显式实现，它指向包含字符向量的内存动态分配块。成员函数 String::getChar 可以定义删除和返回一个串的最后一个字符。这个成员函数可以不改变对象本身的状态，即指针的内容不改变。但是，对象的逻辑状态被修改了。所以，即使编译允许，getChar 也不应当说明为 const 成员函数。

2.9 指向成员函数的指针

在运行时，存在着多次要改变一个函数的行为的情况。成员函数指针是能够达到此目的的一种方式。我们来考虑一个类的应用，它是在电路中表示一个调用滤波器的组件的类。这个电路由一些电感、电容和电阻组成。这个滤波器是 RLC 的特殊类型的电路，其中 R，L，C 分别表示电阻、电感和电容。共同的问题是如何确定滤波器的干涉灵敏度。当电压突然地瞬间改变时，滤波器的干涉灵敏度就会产生一个输出。我们需要建立一个滤波器行为的对象模型。该类的构造函数要求参数指定为滤波器的电感、电容和电阻以及初始电流和电压。类成员函数通过以时间为函数的电路来计算电流量。该组件超限的行为可以有 3 种不同的方式：强阻尼、弱阻尼或临界阻尼。构造函数参数能够决定该电路究竟采用的是哪种方式。

这 3 种方式中的每一个都用以时间为函数的电流公式来表示。对于强阻尼（overdamped）情况，

$$i(t) = A_1 e^{S_1 t} + A_2 e^{S_2 t};$$

对于临界阻尼（critically damped），

$$i(t) = e^{-\alpha t}(A_1 t + A_2);$$

对于弱阻尼（underdamped），

$$i(t) = e^{-\alpha t}(B_1 \cos\omega_d t + B_2 \sin\omega_d t),$$

其中 $S_{1,2} = -\alpha \pm j\omega_d$，

$$\alpha = \frac{R}{2L},$$

$$\omega_0 = \frac{1}{\sqrt{LC}},$$

$$\omega_d = \sqrt{\omega_0 - \alpha^2}。$$

我们想要这个类给出一个单一 current 函数，而内部行为的不同作为函数的语义。

在 C++中，我们可以说明一个指向成员函数的指针，并使用调用一个函数所提供的已经被初始化的指针变量。成员函数指针必须同时指定该类所包含的成员函数的类型以及函数本身的界面；还要对函数指针进行类型检测。在程序执行期间，我们可以再指定给定函数指向几个不同函数的指针，

其中所有的函数指向有相同参数类型和返回类型的指针。

下面是一些函数的说明和一些用于这种声明的函数指针的说明：

函数说明：	兼容的指针说明：
`int String::curColumn();`	`int (String::*p1)();`
`int String::length();`	
`int String::hash();`	
`char Stack::pop(int);`	`char (Stack::*p2)(int);`
`void Stack::push(char);`	`void (Stack::*p3)(char);`
`int PathName::error(int, const char*...);`	`int (PathName::*p4)(int, const char* ...);`

下面是使用以上说明的一些初始化：

```
p1 = &Stack::length;
p2 = &Stack::pop;
p3 = &Stack::push;
p4 = &PathName::error;
```

下面是如何调用这些函数的一些说明：

```
int main() {
    String s;
    Stack t;
    PathName name, *namePointer = new PathName;
    p4 = PathName::error;

    int m = (s.*p1) ();
    char c = (t.*p2)(2);
    (t.*p3)('a');
    (name.*p4)(1,"at line % d\n", __LINE__);
    (namePointer->*p4)(3,"another error  (% d) in file% s",
        __LINE__, __FILE__);
    return 0;
}
```

我们也可以说明指向类对象数据成员的指针：

```
class Table {
public:
    sort();
    ...
};
class X {
public:
    Table t1,t2;
    ...
};
int main() {
    Table X:: *tablePointer = &X::t1;
    X a, *b = new X;
    (a.*tablePointer).sort();          // a.t1. sort()
```

```
    (b->*tablePointer).sort();        // b->t1.sort()
    tablePointer = &X::t2;
    (a. *tablePointer).sort();        // a.t2.sort()
    (b->*tablePointer).sort();        // b->t2.sort()
    return 0;
}
```

我们所编码的电子电路模型又是怎样的呢？例 2.8 表示用于 RLC 电路灵敏度特征的类。类构造函数的参数包括电阻（r）、电感（l）和电容（c）的值，以及电路中的初始电流量。我们可以创建这个类的对象以及带有 time 参数的 current 成员函数，可以在任何时间点上观察电流量的值。

例 2.8：谐振系统的灵敏度的类特征。

```
#include <complex.h>
typedef double time;
class SeriesRLCStepResponse {
public:
    complex (SeriesRLCStepResponse:: *current)(time t);
    SeriesRLCStepResponse(double r, double 1,
       double c, double initialCurrent);
    double frequency() const {return 1.0/sqrt(L*C);}
private:
    complex underDampedResponse(time t) {
       return exp(-alpha *t) * (b1*cos(omegad*t)+
            b2*sin(omegad*t));
    }
    complex overDampedResponse(time t) {
       return a1*exp(s1*t)+a2*exp(s2*t);
    }
    complex criticallyDampedResponse(time t) {
       return exp(-alpha *t)*(a1*t+a2);
    }
    double R,L,C,currentT0,alpha;
    complex omegad,a1,b1,a2,b2,s1,s2;
};
```

但是，“成员函数”current 不是一个函数的全部，而函数指针是由构造函数建立的。构造函数采用电路提供的参数来为灵敏度函数作出决定，而为 current 安排指向了正确的函数。下面是构造函数的代码，其中有初始化 current 函数的指针：

```
SeriesRLCStepResponse::SeriesRLCStepResponse(
   double r, double 1,double c,double initialCurrent) {
   R = r; L = 1; C = c; currentT0 = initialCurrent;
   alpha = R/(L+L);
   omegad = sqrt(frequency()*frequency() - alpha*alpha);
   ...calculation of a1,b1,a2,b2,etc...
   if (alpha <frequency()) {
       current =
         &SeriesRLCStepResponse::underDampedResponse;
   } else if (alpha >frequency()) {
       current =
```

```
            &SeriesRLCStepResponse::overDampedResponse;
    }else {
        current =
            &SeriesRLCStepResponse::criticallyDampedResponse;
    }
}
```

下面是使用 RLC 电路类的简单应用程序：

```
int main()
{
    double R,L,C,IO;
    cin >>R>>L>>C>>IO;
    SeriesRLCStepResponse aFilter(R, L, C, IO);
    for (time t = 1.0; t<100; t + = 1.0) {
        cout<<(afilter.* (afilter.current))(t)<<endl;
    }
    return 0;
}
```

处理以上例子的另外一种方式是使用称为 functor（功能元件）的对象，其中，每个函数用一个对象来替代。这种技术依赖于虚拟函数（virtual function）。在 5.6 节，我们将利用这个相同的例子来进行详细的描述。

2.10 程序组织的习惯约定

通常，类说明按惯例放在头文件中，这个头文件是个 C++源文件，其名字有一个.h 的后缀。该文件有一个与类名相同的清晰的名字，对 Stack 类来说，该文件为 Stack.h。这里，术语“说明”隐含了用户对类的感兴趣的事情，在这些感兴趣的事情中也包含了类的实现。该类的实现按惯例放在有后缀.c 的 C++源文件中（例如 Stack.c）。

类的说明的次序首先是 public 部分，它是很重要的，也是大多数人最关注的部分。然后是用 protected 说明的部分。最后是该类界面的 private 部分。内联（inline）函数一般发生在界面中，并保留在一个头文件中，使用 inline 保留字放在类说明之后。我们应该把内联函数体从它们整齐的类界面中分离出来。为了优化分离，内联函数定义可以放在自己的#include 头文件中，或者可以放在类头文件的底部，或者放在应用代码中。这种组织可以很容易地把已存的非内联函数变更到内联函数之中，反之亦然。整个头文件被包装在某些条件编辑命令中，以便帮助管理员在#include 中进行嵌套：

```
// Stack.h
#ifndef  _STACK_H
#define  _STACK_H 1
class Stack {
public:
    Stack();            //  一个构造函数
    Stack(int);         // 另一个构造函数
    ~Stack();           // 析构函数
    long top()const;
```

```
    long pop();
    void push(long);
private:
    long *items;
    int sp;
};
inline void Stack::push(long i) {
    items [++sp] = i;
}
#endif  _STACK_H
```

练　习

2.1　使用 C++模板（template）机制，重写 Stack 例子，使之能够创建任何类型的栈。

2.2　扩展 Stack 类，使当

```
#include <stdio.h>
...
Stack interactiveStack = stdin
```

时，将创建一个 Stack，从 stdin 流（或任何这样指定的流）中提取它的元素，直至搜索到文件结束为止。

2.3　UNIX 打开子程序用 const char *指定打开文件的名字来扩充。请为一个库草拟一个如何创建头文件的说明和编码，使之能够做到：

```
String group("/etc/group");
FILE *f = fopen("input", "r");
int fd1 = open("/etc/passwd", O_RDONLY);
int fd2 = open(group, O_RDONLY);
int fd3 = open("outfile", O_CREAT,0640);
int fd4 = open(fd2,O_NDELAY);
            //(hint:fcntl(2),fcntl(5))
int fd5 = open(f,O_RDONLY);
```

2.4　使用预处理宏，我们可以对一个 String 抽象数据类型的实例，定义一个将添加.c 后缀的、有效的内联函数，它们是：

```
//+已经当做串连接操作符来定义，如
#define addsuf(s)(s+".c")
```

（1）定义一个做相同事情的 C++内联函数；

（2）使用两种方法来编写一个小的 C++程序；

（3）在程序上添加第二种调用的宏版本，并观察程序大小中增长的字节数；

（4）用第二种内联函数替代第二种宏。注意，该程序的新的尺寸变化。解释两种尺寸递增之间的不同。

2.5　如果是通过 new 操作符运作的话，或者如果是通过已经预先分配内存块的一个对象来运作的话，那么能告知构造函数吗？为什么能或者为什么不能？

2.6　编写一个高效的栈类，要求对该栈的尺寸不能加以固定。

第3章 具体数据类型

具体数据类型（concrete data type）是程序员可以创建的一些类型。用户可以依据C的基本类型定义这些类型的数据表示，也可以依据用户已经创建的具体数据类型或其他抽象类型来定义这些类型的数据表示。用户可以在这些类型上定义自己的操作，即定义它们的名字以及它们的行为。甚至还可以在这些类型重新定义像+、-、*、/这样的C操作。一句话，用户可以像创建C的内置类型int，char，double那样来创建具体数据类型。

具体数据类型应与类的实现区分开来。具体数据类型遵循一种专门的形式，即使用类成员来扩充C++编译类型系统，使得对任何复杂的抽象，编译都能够产生有效和安全的代码。我们把所调用的这种形式称为传统规范类格式（orthodox canonical class form）。“规范”的含义是为编译所生成的代码给出一个规则框架，而“传统”的含义是这种格式最能使人直接地领会，并由语言本身来支持。这是在C++中最好的方式，它给程序员以高度的灵活性，并避免了用户的一些担心。实现ADT，无须传统类格式的许多不必要的东西，在应用上也不受什么限制。

本章从介绍传统规范格式本身入手，然后描述C++数据隐藏机制，以支持类中的信息隐藏。当构建像复数类型那样的类时，重载对大多数规范格式就是必需的，而且是有用的。用户必须指出如何把现成类型的对象，自动地转换成一个新类的对象。下面第3.1节将描述如何进行这种类型的转换工作。

本章还将讨论动态内存管理上的3个重要方面，这是大多数基于对象程序的关键问题。用引用计数方式来介绍减少拷贝对象花费的习惯用法。描述采用控制系统分配和撤消分配的算法以及数据结构的语言机制。最后，展示对象创建和对象初始化之间的关系，并用实例来加以说明。

3.1 传统规范类格式

传统的规范类格式是最重要的C++习惯用法之一。如果用户在构造类中遵循这种“格式”，那么由类所创建的变量能够被赋值、说明以及像C变量那样作为参数来传递。

下面以类String为例，看看当用户需要创建一个类String时所提供的像length、连接等直接的操作。当然，用户希望定义一个有成员函数的新类，为的是在String上进行这些操作，但界面操作需要比串所指定的操作还要多。作为约定，界面应当有像例3.1那样的模式。这种String试图提供一种数据抽象，隐藏了C语言表示串的细节（即char*）。C的串被嵌入到该类的私有部分，它只能用类操作和成员函数来进行直接访问。让我们逐个地检验一下这些成员函数，看看它们是如何实现的。

String():

当没有提供初始化时，这个构造函数用缺省值来初始化 String 对象，称为缺省构造函数，或者有时称为 void constructor（无返回值的构造函数）。对于 String 来说，缺省动作是创建一个空（null）串（如果是 Complex Number 类的话，它就把缺省值置为 0，依此类推）。其代码是：

```
String::String(){
    rep = new char [1],
    rep [0] = '\0';
}
```

这个构造函数使编译可自动地调用初始化对象的向量成员。例如，这个构造函数初始化下面的向量的每个成员：

```
string String Vec[10];    // 每个初始化都有 String::String()
```

如果用户没有定义类的构造函数，那么编译将提供一个缺省的构造函数。上面提到的那个构造函数，将安排调用所有类成员它们自己的类对象的缺省构造函数（在我们这个 String 的例子中，不存在这样的成员）。但是，如果编译提供了缺省构造函数的代码，则没有缺省构造函数的成员将不被初始化。

例 3.1：使用传统规范格式习惯用法的 String。

```
class String {
public:
    // String 的公共用户接口
    // 重定义 "+" 的含义是连接两种情况：
    friend String operator+ (const char*, const String&);
    String operator+ (const String&)const;
    int length() const;                    // 串在字符中的长度
    ...                                    // 其他感兴趣的操作
    // 已特定的成员函数：
    String();                              // 缺省构造函数
    String(const String&);                 // 构造函数从已存的一个串中
                                           // 初始化一个新串
    String& operator = (const String&);    // 赋值
    ~String();                         // 析构函数
    /*
     * 这些操作符是用户能够对一种类型定制行为的常用种类。
     * 这些例子适合 String 类。
     */
    String (const char*);                  // 从 "C 串" 中初始化
private:
    char *rep;                             // 实现数据和内部函数
                                           // （这里，表示的内部就如同旧的 C 串一样）
    };
```

String(const String&):

给出一个已经构造好的 String（并且很可能修改过的），这个构造函数应建立一个精确的拷贝，其参数是一个对 String 的引用。这里，刚才的方法是，这个函数有实际的对象与之工作，即替代刚才给出的指向函数的指针或者拷贝构造函数。

```
String::String(const String & s) {
    // 给'\0'留出地方（位置）
    rep = new char [s.length()+1];
    // 把旧的一个数据拷贝到新的数据之中
    ::strcpy(rep, s.rep);
}
```

其中::strcpy 是引用出现在全局作用域中的那个名字的函数。通常，这个构造函数必须把现存对象的每个数据项拷贝到新的对象的相应的地方，我们将调用这个“拷贝构造函数”，其含义是用拷贝相类似的对象数据项的方法来初始化一个对象。这个“拷贝构造函数”的术语有些使人误解，因为它经常做的实现是逻辑拷贝，或者称对象的“浅拷贝”（浅拷贝所拷贝的对象，不是在物理上拷贝的对象；而深拷贝的含义是递归地拷贝低层对象）。该拷贝是用这个构造函数进行的拷贝，而不需要物理上拷贝。也就是说，它可以是一种逻辑拷贝，而外部行为上应看做是一种物理拷贝。这在本章中对内存共享和管理策略是很有用的。对于这种构造函数的更精确的描述，可以从现存对象中创建“逻辑克隆（logical clone）”，而不是用“拷贝构造函数”来实施。

当一个 String 用值作为参数传递时，由 C++编译产生这种构造函数的调用，并且有时作为一个值来返回。

String operator = (const String&):

这个赋值操作符（=）很像“拷贝构造函数”，它具有两个小的例外。首先，它有一个返回值，而构造函数没有。如果一个成员函数不是构造函数的话，若调用者希望的话，它必须提供一个返回值。其结果是用赋值操作符返回的大多数适合的类型引自当前类型，以便节省创建中间或暂时对象的开销（附录 D 包含了有关赋值操作符的进一步的讨论）。

第二，因为这个对象的内容是用一些新的内容（其中有赋值方法）来替代，这个函数有销去旧的对象的义务。赋值操作符必须显式地撤消内存，即撤消对新值的任何方式的引用。因为使用规范格式，所以保证在 operator = 之前调用构造函数，它给出所有构造函数应做的工作，并恰当地初始化 rep。因此，在赋值期间就知道撤销了旧的 rep。

```
String & String::operator=(const String & s) {
    if (rep != s.rep) {  // 专门检查 a=a
    // 下面删除旧的 rep:方括号告诉编译器要
    // 删除一个向量
       delete[]rep;
    // 好像 String::String (const String &)一样
       int lengthOforiginal = s.length()+1;
       rep = new char[lengthOforiginal];
       ::strcpy(rep,s.rep);
    }
// 必须返回此结果，而构造函数做不到
// 这个结果是对此对象的引用
return *this;
}
```

变量 this 是指向“当前”对象的内部变量。编译自动地提供给用户，而且不需要做什么说明。变量 this 是 C++保留字，当然不能作为普通的变量名来使用。它的类型是指向某些类的一个对象的常数指针，特别情形是类包含着成员函数。注意，绝不能对变量 this 赋值。

重定义赋值的语义是重载。我们在对 String 给出赋值操作符的含义时，对其他类型赋予的含义是不同的。缺省的类对象赋值的含义是把右边对象堆栈的一个个成员赋给赋值操作符左边的相应的堆栈之中。

operator = 的说明产生返回一个成员类型的引用。表达式 a=b 本身的含义是左边赋值，这样的赋值表达式称为左值（l-value）。下面介绍一个合法的表达式会产生副作用：

```
String a,b,c;
(a = b) = c;
```

这样一个语句的语义是迷惑不清的，但我们仍可能要强制使用它。替代赋值的另一种方式可以是

```
const String & String::operator = (const String & s) {
        ...
}
```

第一个 const 修饰符防止用其他成员函数来修改该表达式所计算出的结果，使用左关联的赋值是非法的。使用潜在构造函数将付出一定的代价，诸如

```
(a=b).put('C');
```

是非法的。

~String( ):

这个析构函数必须撤消该对象用它的构造函数所获得的资源，或者是用它的成员函数一起所获得的资源。String 的其他成员函数必须记录在 String 的成员数据的动态分配资源上，所以析构函数能够找回这些资源。这里，仅动态分配的数据项是指向 rep 的 C 串。再者，因为我们使用了规范格式，所以知道调用的构造函数也一定被析构函数所调用，即知道 rep 指向在堆上的串，其代码也是很简单的：

```
String:: ~String(){
   delete[]rep;
}
```

注意，析构函数从来没有参数，任何一个类至多有一个析构函数。

String(const char*):

这个构造函数从一个 C 串中创建了一个 String。它是一个指定 String 的构造函数，并且大多数类型有它们自己定制的构造函数，以便初始化其实例。构造函数可以有多个参数，一个类可以有几个具有不同参数组合的构造函数，编译能够通过语义自动地、正确地选择其中一个构造函数。这个构造函数的代码为

```
String::String (const char *s) {
   int lengthOfOriginal = ::strlen(s)+1;
   rep = new char[lengthOfOriginal];
   ::strcpy(rep,S);
   }
```

这本来不是传统规范格式的一部分，但它是普通用户提供的构造函数，其语义与应用有关。此外，它是用构造函数初始化 String 的，当提供的一个字符指针期望指向 String 对象时，它也是个定

义如何构造 String 的转换操作符：

```
extern int hash(String);
...
hash("character literal");
    // String::String(const char*) 被自动地调用
    //（并且或许是 String::String (const String &)）
```

length()：

length 操作符也是对 String 的指定，并有一些简单的代码：

```
int String::length() const{
    return::strlen(rep);
}
```

这本来也不是传统规范格式的一部分，但它是普通的成员函数，其语义与应用有关。

这个 String 例子描述了习惯用法和模式，然后就可以实现类。对某些类 X 来说，这种格式具有如下特征：

- 缺省的构造函数(X::X())；
- 拷贝构造函数(X::X(const X &))；
- 赋值操作符(X & operator=(const X &))；
- 析构函数(X:: ~X())。

当使用这种习惯用法时，如果

- 要支持类对象的赋值的话，或者要把这些对象作为用值参数调用来传递给函数的话，以及
- 该对象包含指向引用计数的对象的指针，或者类析构函数在该对象的数据成员上执行 delete 的话，

那么，通常就必须使用传统规范格式。

对程序中任何非普通类来说，就应该使用传统规范格式，此格式能够保持交叉类之间的一致性，并能够管理程序演变中的每个类所增长的复杂性。

此外，本书后面的习惯用法将建立在传统规范格式的基础上。若对应用程序提出一定的建议，那就应当是使用这些习惯用法。

在面向对象 C++程序中，指针是普通的，而大多数类将遇到上述的准则。鉴于一致性原则，大多数非普通类应遵循传统规范格式。“普通”类在这种上下文中，对数据聚集来说，习惯上作为 C 的结构（struct）来使用。如果成员函数习惯用于控制对类的数据的访问，并且如果这些数据包含了指针，那么单凭经验的做法就应使用传统的规范格式。

规范格式中的某些变化单用选择缺省行为方式是无能为力的。使用缺省的构造函数 private 创建这个类的实例是不可能的，特别是这个类未提供指定具体构造函数的参数的时候。拷贝对某些类是无意义的。例如，在微处理器中，对象的成员是把成员所在的内存映射到 I/O 寄存器，在这种情况下，不应该拷贝也不能移动。拷贝图形系统的 Window 对象也是没有意义的。如果拷贝 Window 对象是合法的话，那么它可能用值传递给函数，而每个函数调用具有值 Window 参数，因此会在屏幕上出现一个新的 Window!使用这些类的 operator=和拷贝构造函数 private 以防止这些类的对象拷贝。

在面向对象设计中，习惯使用的一个公共的范畴是包容器（container）类，其对象类型设计成

可容纳其他类的集合。包容器类的例子有集合、列表、队列、（可运行）程序库、无序的单位组和栈。包容器类应使用这种格式或其他变型。

传统规范格式的基础是具体数据类型，其他规范格式在本书后面将会建立。

3.2 作用域和访问控制

除了类自己的成员函数以外，类还能够控制它的成员对函数的访问，按照惯例，这种访问控制称为“横向访问控制”。与此相反的是“水平访问控制”，水平访问控制与基类和导出类之间的访问有关。完全的水平访问控制管理需要实施封装性，以便确保类在使用时语义的合理性以及用于客户端作法（方式）的一致性。水平访问控制也能帮助用户在复杂系统中支持抽象，支持用户定义类型的应用的重要表示。我们在第 5 和第 6 章将考察访问控制、继承和面向对象程序设计之间的相互作用。

注意，在例 3.1 中，类 String 有两个标号，一个是 private；另一个是 public。在“public:”之后，可以直接访问任何 C++函数。如果函数有一个指向 String 的指针，那么可以使用指向调用任何一个函数的指针，或者可以访问任何带有 public 标签的数据。private 成员（“private:”标号之后）仅能用它的类的成员函数进行访问，这个类外面的私有成员是不能直接访问的。这里存在一个例外，即 friend（友元）机制，它允许其他成员函数违背类的封装原则。友元的说明格式为

```
friend class George;
```

该类 George，授权给所有成员函数都可以访问归属说明中的类的 private 成员和 protected 成员，例如，

```
class Sue {
   friend class George;
public:
   ...
};
```

Sue 可以窥视到作为 friend 的 George 的说明，即使在上例中也看不到什么。George 允许阻止实现 Sue 的封装，其说明格式为

```
friend Sally::peek(int);
```

有更大的局限性，它只有在给出类 Sally 命名的成员函数时才能访问当前类的私有成员。

私有成员函数对类的实现隐藏算法是有用的，它未曾宣布是类界面的一部分。

成员函数可以访问它们自己的类的任何成员（private 或 public 的成员）。一个基类的保护成员可用导出类的成员来访问，这仅当它们是导出类的组成的一部分并且按此方式来访问时才成立。

作为缺省，struct 行为就好像是一个类的所有 public 成员：

C 代码：

```
struct Stack {
   ...
};
```

C++代码：

```
class Stack {
public:
   ...
};
```

但是，struct 也可以指定它的成员的保护：

struct 代码：

```
struct Stack {
    void push(int);
    int pop();
    int top()const;
private:
    int rep[100];
};
```

class 代码：

```
class Stack {
public:
    void push(int);
    int pop();
    int top() const;
private:
    int rep[100];
};
```

尽管私有数据能够被该类的成员函数访问（读和写），但成员函数可以限制对类的外部用户的访问（如读）。例如，我们可以用 C 语言的上下文来表示 String（见例 3.2）。这样比建立 String 内部数据拷贝（使用该类外部）的效率会更高一些。但如果滥用或私自改变 String 的内部状态，则将是很危险的。

例 3.2：一个类公开它的内部数据。

```
class String {
public:
    String();                                   // 缺省构造函数
    String(const String&);                      // 拷贝构造函数
    String& operator = (const String&);         // 赋值
    ~String();                                  // 析构函数
    String(const char*);                        // 从“C 串”中创建 String
    String operator + (const String&)const;
                                                // 重定义+的意思是连接
    int length()const;                          // 串在字符中的长度
    const char *const C_rep() {
        return (const char *const)rep;
}
private:
    char *rep;
};
int main() {
    String s;
    ...
    printf("string is % s\n",s.C_rep());
}
```

这里，用引用更改了语法：

```
class String {
public:
    ...
    String(const char *s):C_rep(rep) {
        rep = new char[::strlen(s)+1];
        ::strcpy(rep,s);
    }
    const char* &C_rep;
    ...
```

```
private:
    char *rep;
};
```

3.3 重载：重定义操作和函数的语义

我们已经描述了当保持赋值高层语义时，如何控制赋值的操作。当然，我们也能够执行像大多数 C 操作一样的事情。重定义操作（称为操作符重载）对用户定义类型是很重要的。我们根据在用户定义类型上的操作赋予其特征类型。考虑 C 语言所支持的操作类型，如算术类型，有+，-，*和/操作。按照惯例和通常的理解，我们确实要控制这些操作，因此采用从用户定义 Complex 类，Vector 类和 Matrix 类的方式来处理这些操作对象。C++超越 C 的一大优点就是用户定义类型的透明性，诸如 Complex，它们可以像 int 和 double 一样使用。在 C 中，甚至像 typedef 都并不能真实地建立一个新的类型，因为它并没有影响到如何执行操作（如+和-），C++具有做这些工作的能力。本章将描述具体数据类型的作用和操作符重载的作用。

重载的另一个潜在的问题是与算术类型实例相比缺少一定的清晰性。尽管如此，重载还是有用的。例如，我们已经指定有+操作的 String 类，并希望具有串连接的语义，所以左边的代码被解译成与右边的代码具有相同的含义（在例 3.2 中，C++给定的说明都有效）。

```
int main() {
    String a,b("this is b");
    String c("!");
    a = b + c;
}
```

```
int main() {
    String a,b("this is b");
    String c("!");
    a=b.operator+(c);
}
```

因为我们无论要做什么，总可以定义 operator+，当用于 String 时，我们可以定义 operator+做与 String 的连接的动作，其代码是：

```
String String::operator + (const String& s) const
{
    char *buf = new char[s.length() + length() + 1];
    ::strcpy(buf,rep);
    ::strcat(buf,s.rep);
    String retval(buf);
    delete[]buf;            // 删除临时存储区
    return retval;
}
```

注意，该函数创建了一个新的 String，它实际上调用了 String::String(const char*)，从 buf 中构造了 String 对象，并且这个对象成为 operator+的局部变量。然后，从 operator+返回一个新对象的拷贝，此拷贝实际上是与调用 String::String(const String &)的构造有关的拷贝构造函数。

这里涉及到使用重载的一些风格。在类上重载 operator[]就好像向量（vector(i)）做那些合法的事情一样。重载 operator=对规范格式是必须的。其他的使用，诸如重载算术操作符含有非算术的语义，但在可用性上存在某些争论。如果这些操作符习惯在某些上下文中被引入使用（例如，在串上的 operator+可以有连接的含义）的话，那么限制这些操作符的使用可以改善其灵活性，但也不能滥用。

一个重载下标的例子

我们甚至能够重载表面上不大可能作为下标的操作符。下面我们从能够有下标的 String 中挑选出 String 的单个特征来。在类说明中添加这个操作符的一个说明，

```
class String {
public:
    ...
    char & operator[](int);
    ...
};
```

我们对该函数本身进行编码，其代码是：

```
char String::operator[](int index)
{
    return rep[index];
}
```

如果我们希望的话，这个函数可以被当做作用域来检测，以确保 index 是正的，并且比 length() 要小。它可以打印错误信息，如果检测到有关条件则可以返回 EOF 字符（文件结束字符）。使用这些构造，用户可以添加错误检测码，用 String 来调试应用程序而无须非得改变整个应用程序。而一旦程序被调试，故障诊断码就可以删去。更好的做法是，在源程序中舍弃故障诊断码，而用#ifdef 来替代，使故障诊断码免于编译，直至此码再一次被需要为止。

因为 operator[]的返回类型被说明为对字符的引用，所以它可以在赋值的左边使用。有关引用的详细信息，参见附录 A.9 节。

用户不一定对整数自变量采用重载 operator[]；operator[]可以采用任何类型的自变量，或者能够为几个不同的类型重载多次。如果有多重说明的话，编译可从调用的上下文中作出正确的决定。例如，我们可以对用 const char*参数的 operator[]重载下标：

```
some_type operator[](const char*)    // 关联数组
```

这习惯用于聚集对象的成员操作，诸如列表和无序单位组（bag），它们是用名字来检索对象。我们也可以对 String 添加这样的成员函数，以便查找表示子串的实例。

更进一步的例子，应考虑重载 operator[]，但其实现与使用左值（l-value）和在给定的索引上使用检索值之间可能有些区别。我们使用的是基于 operator[]版本的一种技巧，即创建一种虚拟类型的临时返回值。当创建虚拟对象时，或当出现在某个上下文中时（例如，作为赋值的左边，或者在所预料的另一类型处的上下文中），它自己重载的操作符用控制产生所预料的结果。例 3.3 给出了 File 抽象的一个例子。

例 3.3：有上下文语义的重载 operator[]。

```
class FileRef {
private:
    class File &f;
    char buf[1];
    unsigned long ix;
public:
    FileRef (File &ff, unsigned long i):f(ff),ix(i) {}
    FileRef &operator = (char c);
```

```
    operator char ();
};
class File {
friend class FileRef;
public:
    File(const char *name) {
       fd = open(name, O_RDWR | O_CREAT,0664);
    }
    ~File() {close(fd);}
    FileRef operator[] (unsigned long ix) {
       return FileRef(*this, ix);
    }
private:
    int fd;
};
FileRef& FileRef::operator = (char c) {
   lseek(f.fd, ix, 0); write(f.fd, &c, 1); return *this;
}
FileRef::operator char() {
   lseek(f.fd, ix, 0); read(f.fd, buf, 1); return buf[0];
}
int main() {
    File foo("f00");
    foo[5] = '5';
    foo[10] = '1';
    char c = foo[5];
    cout  <<"C="<<c<<endl;
}
```

当 File 对象用左值来索引时，需要重写该文件的指定字节。否则，就要推导出该文件的指定字节，并作为表达式的值来返回。

在 File 对象上引用 operator[]产生了隐蔽的“虚”类型 FileRef 的结果，此对象既可以作为左值来赋值，又可以直接用它的值来使用，对 FileRef 对象的左边使用 File::operator[]。现在 FileRef 对象在左边，而字符在右边，FileRef::operator=是在写那个地方被引用。当 File 对象作为下标时，该下标将再一次计算中间 FileRef 对象，然后用 FileRef::operator char 把此对象转换成 char。读该文件的编码可以放在那个转换操作符中，这样，当下标操作的结果作为赋值的目标使用时，就做写的动作；而当下标操作的结果作为其值来使用时，就做读的动作。

成员操作符重载与全局符重载

在 C++中有两种类型的重载。一种是上述例子中所给出的成员重载；另一种是全局重载，它是在全局作用域层面上来实施的。假定 *a* 和 *b* 都被说明为类 C 的对象，类 C 用 C::operator+(C)定义，这是一种成员函数重载的情况，类似于前面的例子。在这个程序中，*a*+*b* 的意思是 a.operator+(b)。另一个选择是，我们可全局地重载+操作符：

```
C operator + (c, c) {...}
```

同时，也可应用于 *a*+*b*，分别为该函数提供 *a* 和 *b*，作为第一个和第二个参数。在这两种格式中，我们宁愿使用成员重载。采用第二种格式要求使用 friend（友元），这种格式可能必须在对象代码库中调节类，其中源程序不能修改和重新编译。因此，应当避免把两种格式混合在一起使用。如果两

种格式出现在给定的操作符中，并具有类似的形参类型，那么这个操作符的使用可能会混淆不清，而且有可能导致致命的问题。

但是，当我们必须使用对称的全局重载操作符时，就要考虑一定的美学观点。下面我们来考虑一下多重、全局的重载 operator+函数以及类 String 的所有友元的情况：

```
class String {
friend String operator+ (const char*, const String&) const;
friend String operator+ (const String&, const char*) const;
public:
    ...
private:
    char *rep;
};
String operator+ (const char *s, const String& S) const {
    String retval;
    retval.rep = new char[::strlen(s) + S.length()];
    ::strcpy (retval.rep, s);
    ::strcat (retval.rep, S.rep);
    return retval;
}
String operator + (const String& S, const char *s) const {
    String retval;
    retval.rep = new char [::strlen(s) + S.length()];
    ::strcpy (retval.rep, S.rep);
    ::strcat (retval.rep, s);
    return retval;
}
```

使用这些说明，同时做以下“加法”工作：

```
String s1;
"abcd" + s1;
s1 + "efgh";
```

做这种工作没有全局符重载是不可能的。因为我们不能得到“内部”的本地 C 串类型（即 char*）来定义它的操作；我们也不能用定义类操作的方法支持简单的对称，即全局 friend 函数方法是必须的。当两个允许的自然类类型（与之相反的类型，如 int，char 等）要对每个其他私有成员访问时，友元的关系能够在这些类之间作适当的安排；全局的 friend 函数通常不是必须的。friend 转换函数也可能需要在两个类之间（其中一个类是在库中）进行平滑的转换，而且除了其类的界面说明可以使用之外没有别的源程序可以被使用。例如，我们可能要变更 PathName 类，以便使+操作符能够用于添加 String 的后缀。如果 PathName 是在一个库中（源程序不能用），则全局重载必须使用：

```
PathName operator + (const PathName &p,const String &s)
    {...}
```

3.4 类型转换

借助于添加的构造函数和操作符的定义，我们可以让编译获得足够多的信息，以实施自动类型

转换。例如，根据一致性原则，我们要做以下工作：

```
#include <string.h>
extern"C"int strlen (const char*);
int main() {
    String s("1234");
    int j = ::strlen(s);    // 与 j = s.length()相同
    ...
}
```

为了实施这种工作，我们需要告知编译如何创建“基本”（非类的）类型，这是可以用操作符规范说明来实施的。这里，我们要求编译自动地将 String 转换成 char*，所以在 String 上定义一个操作符去完成：

```
class String {
public:
    operator const char* () const {return rep;}
    ...
};
```

第一个 const 表示指向返回值的内存不应该被该类外部的代码修改，而第二个 const 表示这个成员函数不可以更改操作在 String 对象上的内容。现在，当编译在 const char*有意义的上下文处看到 String 时，它将自动地调用 String::operator const char*以便完成这种转换工作。这个操作符恰好返回指向内部数据的指针，此数据是在 C 语言子程序中所要求的那种格式。如果转换必要的话，就能够把转换放到该操作符的主体中。我们称此为成员操作符函数的习惯用法。

另一个例子是，每当 String 出现时，它的译码串似乎是由整数的 ASCII 来表示，其代码是：

```
class String {
public:
    ...
    operator int();
    ...
};
String::operator int()
{
    int retval;
    if (sscanf(rep, "% d", &retval)! = 1) return 0;
    else return retval;
}
```

为了得出这个函数，String 必须出现在可以解释成整数的上下文中。类型转换既可以是隐式的，也可以是显式的：

```
#include "String.h"
int main()
{
    char buf[10];
    printf("enter number of bytes:");
    scanf("% s \n", buf);
    String sbuf = buf;
    // 显式地对 int 进行转换
```

```
    printf("read in value % d\n", int(sbuf));
    char *thing = new char [100];
    // 显式地对 const char *const 进行转换
    ::strcpy(thing, (const char *const) sbuf);
    // 隐式地对 int 进行转换
    int charsLeftInThing = 100 - sbuf;
    ...
    return 0;
}
```

当使用这种习惯用法时：使用这种习惯用法仅对内置 C 类型进行转换。如果必须把 String 转换成某些其他的类，诸如转换成 ParseTree，那么该转换应当用 ParseTree 来处理，而不是用 String 处理。

为了把 String 转换到 ParseTree，ParseTree(const String &)构造函数应当在可能的情况下对 ParseTree 类进行添加，String 不可能知道每个其他的类。为了使 ParseTree 自由地访问 String 内部，就需要有效地转换，需要在两个类之间建立友元的关系（对像 String 这样简单的类来说，这不可能是一个问题，但它更多的是引用数据结构和类型）。

转换操作符应在适当的设计考虑之后才被引入，过度地使用转换操作符会产生组合代码爆炸，并且也会增加转换二义性的危险。

对于这些经验有三点例外，即做什么事情都应当有意义。那么，我们再来看看所提供的从 String 转换到 ParseTree 的例子。如果在库中提供了 ParseTree，并且只在对象代码和头文件中包含了其类的说明，那么可能难以处理，或者不可能改变 ParseTree 类的内部代码。首先，我们根据类出现在说明中的私有成员这一情况可知，不可能充分地知道有关 ParseTree 内部表达这种转换。为此，我们建议转换必须根据该类的公有（public）界面，它可以在该类的外部来完成。第二，改变 ParseTree 界面无须重写它的库代码，否则会导致“旧”和“新”的 ParseTree 代码之间的不一致性（会产生两种不同的界面）。最后，根据类的公共界面来改变该类的外部，在使用相同的应用程序代码时，可以减轻对新的类库释放的变迁。我们用修改 String 的方法来处理这种转换：

```
#include "ParseTree.h"
class string {
public:
    ...
    operator ParseTree() {...}
    ...
};
```

我们已经看到如何把某些已存的类型转换到 String 的情况。一个例子是 String::String(const char*)，从 const char*中建立一个 String。添加的构造函数能够使任何类型转换到 String。例如，考虑一个 Node（结点）类，它表示字符串二叉树中一个元素：

```
class Node {
public:
    Node();
    Node(const Node&);
    Node& operator = (const Node&);
    Node *leftChild() const;          // 不修改对象
```

```
    Node *rightChild() const;        // 不修改对象
    char *contents()const;           // 因为它不修改此对象，所以就
                                     // 产生了结点的内容说明 const
private:
    ...
};
```

例 3.4 表示了一个构造函数，它把在 Node 下的树拉平到花括号构造之中。有关该构造函数的另一种方式是如何根据 Node 来构造 String，需要使用辅助函数 nodeWalk 来遍历这个树。

例 3.4：把 Node 拉平到 String 的构造函数。

```
#include "Node.h"
class String {
public:
    ...
    String (Node);
    ...
private:
    ...
    String nodeWalk(Node*);
};
String
String::nodeWalk(const Node* n) {      // 嵌入 node walk
    if (n == 0)return"";
    String retval = String("(") + nodeWalk(n.leftChild());
    retval = retval + " " + n.contents();
    retval = retval + " " + nodeWalk(n.rightChild()) +")";
}
String::String(Node n)
{
    String temp = nodeWalk(&n);
    rep = new char[temp.length() + 1];
    ::strcpy(rep, temp.rep);
}
```

3.5 引用计数：使用“可调内存”变量

与过程程序相比，面向对象程序更为经常地把变量与堆栈分配的对象进行捆绑，以替代这些静态说明，诸如全局的、自动的和类成员。这是最基本的事实，因为对象的实例及其数据的生命周期是作为栈结构的对象范型，对过程程序设计来说，它们是活动的记录。在对象中，其大部分数据是可用的，而它们的来、去应当对应于真实世界实体的生命周期。内存分配和回收是个重要的问题，因为程序员可以控制该对象在什么时候创建，什么时候消亡（通过构造函数和析构函数），所以可以引入优化机制以改善类的性能和特殊类的内存使用。

最普遍的机制之一是表示共享和引用计数。这种习惯用法可以对任何 C++类进行概括，特别是对这些类的动态分配内存有利。让我们再来看看上一节中所介绍的 String 类，赋值引起了对整个串的复制，几乎需要双倍的内存要求，以拷贝和储存冗余的信息。在此，我们可以对类添加“灵巧”

的方法，以便替代拷贝，用多个变量共享同一个内部存储区。表示本身是一个对象，它管理着共享数据和有关联的引用计数。这种工作有几种不同的方式，这里介绍3种变体。第一种方式是被大多数抽象所采用的通用形式；第二种方法给出了如何消除用第一种方法引入的某些效率不高的内存拷贝；最后一种方法是寻找如何自动调节这种抽象维护工作的单调性。总之，本节将表示如何把已有的类转换成使用引用计数。

处理类的习惯用法

在一个包含主要作为存储管理动作的类中，首要的技术是封装原始的 String 类（见例 3.1）。我们称这个类为处理类，因为它用于访问另外的类，其中含有应用智能能力的类。该处理由用户作为实际类来实施，但工作是在类的内部进行。所以我们必须做以下几方面的事情：

（1）把类 String 重新命名为 StringRep；

（2）把引用计数域添加到 StringRep 中；

（3）创建一个包含指向 StringRep 的指针的类 String；

（4）把 String 类的大部分操作转送到 StringRep 中，当执行赋值时监视表示的共享。

图 3.1 表示了好像是在内存中的新的 String 对象。命名为 *a* 的 String 对象，初始化常数为“hello world”，而 *b* 赋为 *a* 值。每当对象用 *a* 和 *b* 命名之后，我们就能够区分 String 对象，它们共同指向包含了大部分低层数据和智能串操作的共享表示。该表示带有引用计数，其整数表示有多少个 String 对象的指针指向它。当计数为零时，该对象被释放。

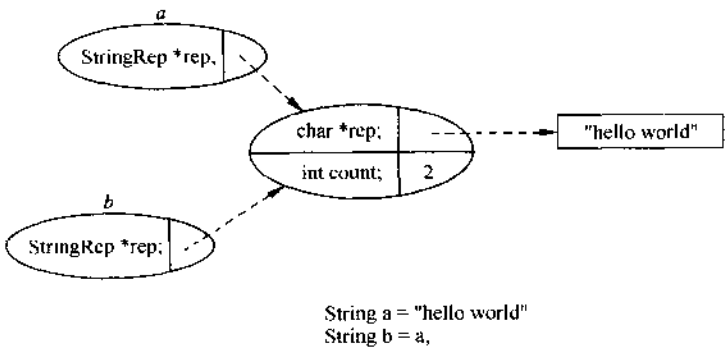


图 3.1 带有引用计数的处理类 String

例 3.5 表示了新的 StringRep 类的说明。除了这里已经移动到内联和构造更加精确的成员函数之外，它的成员函数与以前的 String 是相同的（如果仅在性能分析中作此建议的话，那么在给定的应用中应当构成内联函数。对某些应用来说，生成的代码尺寸可能会产生不可能接受的花费）。

例 3.5：String 表示的一个类。

```
class StringRep {
friend String;
private:          // 现在，只能够对 String 进行存取（访问）
    StringRep()                          {*(rep = new char[1]) = '\0';}
    StringRep(const StringRep& s) {
    ::strcpy(rep = new char[::strlen(s.rep) + 1], s.rep);
    }
```

```
    StringRep& operator = (const StringRep& s) {
      if (rep != s.rep) {
        delete[] rep;
        ::strcpy(rep = new char[::strlen(s.rep) + 1], s.rep);
      }
      return *this;
    }
    ~StringRep()                          {delete[] rep;}
    StringRep(const char *s) {
      ::strcpy(rep = new char[::strlen (s) + 1], s);
    }
    StringRep operator  + (const StringRep&) const;
    int length() const                {return::strlen(rep);}
private:
    char *rep;
    int count;
};
StringRep StringRep::operator  + (const StringRep& s) const
{
    char *buf = new char [s. length() + length() + 1];
    ::strcpy (buf,rep);
    ::strcat (buf, s.rep);
    StringRep retval (buf);
    delete[] buf;                // 删除临时存储区
    return retval;
}
```

现在每当处理内存管理本身许多事务时，我们就需要一个新的 String 类，它使用 StringRep 去做所有与串相关的工作。这个类在例 3.6 中给出。

注意，“未处理”的构造函数对一个对象设定表示计数（count）域，正好拷贝另外一个串（String(const String &）和 operator=(const String &))的这些操作借用了原始表示并递增它的引用计数。修改或替换这种表示必须递减当前表示的计数。当它的计数为负时，则返回并释放存储区。

例 3.6：StringRep 的 String 界面。

```
class String {
public:
    String () {
        rep = new StringRep; rep ->count = 1;
    }
    String (const String& s) {
        rep = s.rep; rep ->count ++ ;
    }
    String& operator = (const String& s) {
        s.rep ->count ++ ;
    if ( -- rep ->count <= 0) delete rep;
    rep = s.rep; return *this;
    }
    ~String()          {
```

```
        if( -- rep ->count <= 0) delete rep;
    }
    String (const char *s) {
        rep = new StringRep (s);
        rep ->count = 1;
    }
    String operator+ (const String& s) const {
        StringRep y = *rep + *s.rep;
        return String (y.rep);
    }
    int length() const {
        return rep ->length();
    }
private:
    StringRep *rep;
};
```

这个方法的一大好处是，在内部的类中舍弃了“串”的所有知识，并舍弃了外部类处理大量的内存管理的事务。这种方法无疑是把现存类转换成使用引用计数的好方法。但是，它可能会造成效率严重的低下。例如，operator+在表达简明性方面做了许多冗余工作。在这个代码中，operator+的引用产生了3个堆的分配和可以在栈上建立一些附加的临时对象。如何改善这种状况呢？我们在下节将加以讨论。

我们称String是处理类，并且是用处理来操纵StringRep的主体。如果其中的一个类的实例充当另一个类的实例的管理者，那么这样两个类（潜在的或者更多）的使用，被称为处理/主体类习惯用法。

当使用这种习惯用法时：处理/主体类习惯用法用于把复杂的抽象实体分解为小的可管理的类。这种习惯用法可以用来控制访问多个类映射到单一资源的共享。它的最普遍的使用是引用计数。

正如在String例子中所看到的，处理/主体类习惯用法的最通常的使用是引用计数。当主体类包含了用处理类操纵的引用计数时，这样的使用称为引用计数习惯用法。

当使用这种习惯用法时：特别是在对象很大或者对象很复杂，需要在拷贝上花费很大时，使用引用计数的类对象就很合适，因为这些类中的实例经常通过赋值或参数传递来拷贝。使用这种引用计数虽然数据本身可能不被拷贝，但实际上却存在着对象的多个逻辑拷贝。例如，计算机屏幕对象是在函数之间传递的，或者是在处理之间局部共享内存的，而每个用户希望保存他自己的对象的拷贝。

更好的保存方法

第二种方法彻底地从头开始，这种方法采用实时的“保存”，并能节约开销。在例3.1的原始的String类之后，把由用户直接访问的String类模块化。它（见例3.7）表示在被扩充之前所包含的一个计数域，并通过间接的多个层次之一进行访问。计数和数据指针被组装成一个具有单个简单构造函数和一个简单的析构函数的简单类。

我们能够在一些“后备门”中装上保存String的operator+(const String&)成员函数引用的拷贝。我们创建一个新的构造函数，采用把字符指针当做它的参数的方法来引用，即该指针用引用来传递，构造函数接收它的值，并且空值表示在原来的构造函数外面（见例3.8）。

例 3.7：更有效的 String 实现。

```
class StringRep {
friend String;
private:
   StringRep (const char *s) {
      ::strcpy(rep = new char[::strlen(s) + 1], s);
      count = 1;
   }
   ~StringRep() {delete[] rep;}
private:
   char *rep;
   int count;
};
class String {
public:
    String()                        { rep = new StringRep(" "); }
   String(const String& s)          {rep = s.rep; rep ->count ++ ; }
   String& operator = (const String& s) {
      s.rep ->count ++ ;
      if (-- rep ->count <= 0) delete rep;
      rep = s.rep;
      return *this;
   }
   ~String() {
      if ( -- rep ->count <= 0)  delete rep;
   }
   String(const char *s)           {rep = new StringRep(s); }
   String operator + (const String&) const;
   int length() const              {return::strlen(rep ->rep); }
private:
   StringRep *rep;
};
String String::operator+ (const String& s) const
{
   char *buf = new char[s.length() + length() +1];
   ::strcpy (buf, rep ->rep);
   ::strcat (buf, s.rep ->rep);
   String retval (buf);
   delete[] buf;                   // 删除临时存储区
   return retval;
}
```

例 3.8：优化连接的一些技巧。

```
#include <string.h>
class StringRep {
friend String;
private:     // 因为是私有，所以不会对全局名字空间构成污染
   typedef char *Char_p;
   StringRep (StringRep::Char_p *const r) {
```

```
        rep = *r; *r = 0; count = 1;
    }
public:
    StringRep(const char *s)
        {::strcpy (rep = new char[::strlen(S) +1], s); count=1; }
    ~StringRep () {delete[] rep; }
private:
    char *rep;
    int count;
};
class String {
public:
    String()                {rep = new StringRep(" "); }
    String(const String& s) {rep = s.rep; rep->count ++; }
    String& operator = (const String& s) {
        s.rep ->count ++; if( -- rep ->count <=0)delete rep;
        rep = s.rep;
        return *this;
    }
    ~String() {if( -- rep ->count <=0) delete rep; }
    String(const char *s)  {rep = new StringRep(s); }
    String operator+ (const String&) const;
    int length() const    {return::strlen(rep ->rep); }
private:
    String(StringRep::Char_p* const r) {
        rep = new StringRep(r);   // 调用新的构造函数
    }
    StringRep *rep;
};
String String::operator+ (const String& s) const
{
    char *buf = new char[s.length()+length() +1];
    ::strcpy (buf, rep ->rep);
    ::strcat (buf, s.rep ->rep);
    String retval(&buf);   // 调用新的私有构造函数
    return retval;
}
```

这里，在 String 的 operator+(const String&)中的代码利用了 StringRep 的构造函数 StringRep (char_p* const)所提供的特殊手段。因为在这两个类之间存在着友元的关系，所以这个构造函数是可以被访问的。这两个函数签订了“君子”协议，即它们是借用 String::operator+(const String&)来保存该字符所分配的内存的。一个指针指向传递到 String 的构造函数 String(const char_p*)中，然后再把它转交给特殊的 StringRep::StringRep(char_p*const)构造函数，此构造函数保留了该指针。专用的 StringRep 构造函数致使将在 operator+(buf)中的原始指针调节到零之外，以确保剩余的存储区中没有悬挂引用。所有这些措施都避免了在 String::operator+中构造 retval 对象时拷贝串表示的现象。

计数指针

计数指针技术与先前两种方法有很大的不同。这种习惯用法起源于对通用垃圾收集（garbage

collection）技术的研究，其思想是把指针置入真正的对象中，每次仅拷贝一个指针，然后调用它的operator=，或者调用拷贝构造函数，所以在用多个计数指针共享单一表示对象中，需要跟踪大量未完成的对对象的引用。当一个指针超出了作用域时，它的析构函数就被调用并且引用计数因此而递减 1；当计数为零时，则程序工作块被复原，即或者是在现场复原，或者采用清除子程序的方法来复原（这里用以前的子程序来做）。这个技巧仅限于在对象上使用操作符重载的方法来控制->操作符的引用。

重载 operator->工作与 C++其他重载操作符不同。对所有其他操作符来说，主体定义的操作符的实现具有最终控制此操作的返回值。对 operator->来说，返回值是基于->语义的中间结果，然后加以应用，才产生一个结果，这样，

```
class A {
public:
   B *operator ->();
};
...
int main() {
   A a;
   ... a ->b ...
}
```

其步骤如下：

（1）在对象 *a* 上引用 A::operator->( )；

（2）从暂时的引用（x，类型为 B*）中获取返回值；

（3）计算 *x->b*，产生它的结果。

这里，*b* 可以用任何类 B 的成员（数据和函数）来替代。如果类 B 重载了 operator->，那么以上 3 个步骤再重来一次。

这种方法不仅具有先前引用计数技术的好处，而且提供了进一步开发的可能性。考虑一个程序，其中的计数指针仅是类对象种类（这里，类对象的含义是指对象中的类型是一些类的类型；而其他对象仍为 int，char，vector 类型以及其他在 C++程序中所能找到的 C 类型）。首先，计数指针给具体数据类型带来了极大的好处，即可以作为参数、赋值、创建、清除等来传递。第二，计数指针携带了与正规指针相关的一些特性，诸如从堆栈中动态地分配访问对象。但是，该对象是被实例化的无须使用 new，并且无须使用 delete 便能自动回收资源。即，它们有相同的分配行为（我们期望类对象自动地分配和静态地实例分配）。例如，如果把 String 当做一个计数指针来实现的话，而且 String 嵌在类 A 中，则有如下语句，

```
class A {
public:
   A() :s("hello world") {}
private:
   String s;
};
```

然后，由类 A 创建的每个对象将包含计数 String 指针 s，把 String 初始化为“hello world”。类 A 的一个对象的拷贝明显地包含一个指针，它指向与原始的 String 相同的指针，即我们把拷贝 s 看

做是拷贝一个指针。但是，它仍然具有真正引用计数对象的所有特性。例如，改变其值会导致它的表示变成不共享了。换言之，计数指针就像一个有构造函数的指针，所以，当计数指针进入到作用域中时，就自动地初始化，指向新的分配对象。

例 3.9 表示了如何把 StringRep 作为计数指针来加以实现的过程。这个例子基于前面的例子，但连接的处理是从 String 到 StringRep 而移植过来，与操纵的信息密切相关。所以，从 const char*和 print 操作中构造了 StringRep 的一个构造函数，对添加一些操作是很方便的。

例 3.9：计数 String 指针的“内部实现”。

```
class StringRep {
friend String;
private:
   StringRep()        { *rep = new char[1] ='\0'; }
   StringRep(const StringRep& s) {
      ::strcpy(rep = new char[::strlen(s.rep) +1], s.rep);
   }
   ~StringRep()       {delete[] rep; }
   StringRep(const char *s) {
      ::strcpy(rep = new char[::strlen(s) +1], s);
   }
   String operator+ (const String&) const {
      char *buf = new char [s ->length() + length() + 1];
      ::strcpy(buf, rep);
      ::strcat(buf, s ->rep);
      String retval (&buf);
      return retval;
   }
   int length() const     {return::strlen(rep); }
   void print() const     {::printf("% s\n", rep); }
private:
   StringRep(StringRep::Char_p* const r) {
      rep = *r; *r = 0; count = 1;
   }
   char *rep;
   int count;
};
```

例 3.10 描述了 String，现在没有显式 length 操作。任何对 String 引用 length 的操作用 operator->转送到 StringRep。对其他成员函数也依此进行，如 strchr（查找字符串），hash 等等。类 String 继续执行大部分内存管理工作，并处理在 StringRep 实例内的引用计数。例 3.9 给出连接操作符 StringRep::operator+的实现，它表达了根据 StringRep 成员的原则，保存超出数据间接层次以外的例 3.8 的实现。因为 String 对 StringRep 进行了转送 operator+的引用，所以这个方法带来了对每次 operator+引用的附加函数调用的花费。这种花费通常能够通过使用内联函数扩展的方法而优化掉。

例 3.10：计数 String 指针的公有界面。

```
class String {
friend class StringRep;
public:
```

```
    String operator+ (const String &s) const { return *p +s; }
    StringRep *operator ->() const { return p; }
    String() {
        (p = new StringRep ()) ->count = 1;
    }
    String(const String& s) { (p = s.p) ->count ++ ; }
    String(const char *s) {
        (p = new StringRep(s)) ->count = 1;
    }
    String operator = (const String& q) {
        // 因为变化，所以这时有一点区别
        if ( -- p ->count <=0 && p ! =q.p) delete p;
        (p = q.p) ->count ++; return *this;
    }
    ~String()              {if ( -- p ->count <=0) delete p; }
private:
    String(StringRep::Char_p *r) {
        rep = new StringRep(r);
    }
    StringRep *p;
};
```

现在，不用 String 作为实例，而是用 String 就像使用指针一样（String 仅表示成是一个指针的类，我们将它看做是一个带有操作符的指针）。实际上，现在我们引用 String 也只是通过这些指针。惟一有点奇怪的是，它们不出现在指针说明中：

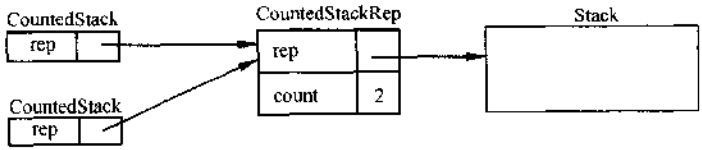
```
int main()
{
    String a("abcd"), b("efgh");
    printf("a is"); a ->print();
    printf("b is"); b ->print();
    printf("length of b is % d\n", b ->length());
    printf("catencation is"); (a + b) ->print();
    return 0;
}
```

计数指针的一大优点是，新的成员函数只需要添加到主体类中。例如，大部分新串操作可以作为 StringRep 的成员函数来实现，即处理类 String 通过它的 operator->自动地得到这些操作。

当使用这种习惯用法时：使用计数指针可以大大减轻人工处理动态内存分配的负担。当开发原型时，计数指针也可以减轻在它的处理类的主体类中复制变动的负担。计数指针的使用主要取决于风格和嗜好，但其仍然是符号习惯用法的重要组成部分（见第 9 章）。

对已存的类添加引用计数

一个已存的类现在可以有更好的选择了，即可以用它的引用计数来改善其性能。但是，缺乏对源文件的访问；限制了类的大小或布局，期望用添加的计数域来倒转；在其他的干扰中，仅在部分时间需要使用引用计数，使得做此项工作遇到了不同程度的障碍。对这样的类可以应用处理类的附加层次。假定类 Stack 在库中已经存在，它作为非引用计数类，并且我们对部分应用需要使用引用计数栈。首先，我们使用已存的 Stack 类创建一个新的栈抽象，命名为 CountedStack：



这里，原始的 Stack 抽象没有涉及，而两个新类由原始 Stack 类管理。类 CountedStack 对付内存管理，而另一方面，CountedStackRep 充当简单的数据结构，添加一个间接层次：

```
class CountedStack {
public:
   CountedStack(): rep(new CountedStackRep) {}
   int pop() { return rep ->rep ->pop(); }
   void push(int i) { rep ->rep ->push(i); }
   CountedStack(const CountedStack &c): rep (c.rep) {
      rep ->count ++;
   }
   CountedStack (const Stack &c):
      rep(new CountedStackRep (new Stack (c))) {}
   operator Stack() {return *(rep ->rep); }
   ~CountedStack() {
      if ( -- rep ->count <=0) delete rep;
   }
   CountedStack &operator = (const CountedStack &c) {
      c.rep ->count ++ ;
      if (-- rep ->count <=0) delete rep;
      rep = c.rep; return *this;
   }
private:
   Struct CountedStackRep {
      int count;
      Stack *rep;
      CountedStackRep (Stack *s = 0): count(1) {
         rep = s?s: new Stack;
      }
   } *rep;
};
```

这里，允许 CountedStack 对象和 Stack 对象交替使用的一些附加工作已经完成。

内存管理和单条（singleton）信件的信封类

我们前面描述了如何把 String 类拆分成两个独立的类，以减少存储区的复制和对象拷贝的开销的问题。外部 String 类是处理类，而 StringRep 类是主体。这两个类对用户是可见的，并且存在于全局（类）名字空间中。包含指向动态分配资源指针的所有类，如果它们是作为具体数据类型而被使用的话，那么应作为处理/主体对或信封/信件对来加以实现。这里，我们展示出多少个主体和信件类可以在它们各自的处理和信封类中加以封装，以构成更紧密的抽象。

String/StringRep 构造是单条信件类的一个例子。信封类，String 被设计成仅保留一种信件种类的 StringRep。这两种构成了一个组合对象，作为一个单一实体来应用，所以不是把信件的实例放到

信封的实例内，而是把信件类放到信封类之中（见例 3.11）。

例 3.11：作为嵌套类封装的单条信封。

```
class String {
public:
   String()                                {rep = new StringRep(" "); }
   String(const String& s)                 {rep = s.rep; rep ->count++; }
   String& operator = (const String& s) {
      s.rep ->count ++ ;
      if ( -- rep ->count == 0)
         delete rep;
      rep = s.rep;
      return *this;
   }
   ~String() {
      if ( -- rep ->count <=0) delete rep;
   }
   String (const char *s)                  {rep = new StringRep(s); }
   String operator + (const String&) const;
   int length() const                      {return::strlen(rep ->rep); }
private:
   class StringRep {
   public:
      StringRep (const char *S) {
         ::Strcpy (rep = new char[::strlen(s) + 1], s);
         count = 1;
      }
      ~StringRep( )    { delete rep; }
      char *rep;
      int count;
   };
   StringRep *rep;
};
```

当使用这种习惯用法时：当信封/信件对用于引用计数时，或者当信件界面只对信封可见时，就应使用单条信件类。这种习惯用法对稳定的抽象是最有用的。

对这种方法的成员函数代码与以前 String 版本的成员函数代码没有什么区别。这种方法改进了设计封装，并且比以前使用的两个全局类具有更少的全局名字空间的“污染”，它的花费随着界面的变化而变化，或者信件类的数据要求重新编译更多的代码。

多个信件类可以嵌套在给定的信封之中，第 5 章将介绍一个简单记号语法分析类，习惯用于产生原子记号对象。这个语法分析类 GeneralAtom 是为信件类（诸如 Punct，Name 和 Oper）而设计的信封类。与 StringRep 类不同，它对用户是不可见的，这些信件类可以直接用于应用。该信件能够在 GeneralAtom 信封的公共界面中加以说明，这样的嵌套类必须是用修饰符命名的语法（如 GeneralAtom::Punct）访问。如果在信封类中有太多的信件类的话，会导致信封类界面变得复杂化，理解上会很困难。这些信件类的应用可派生出信封类（如 Complex，BigInteger 和 DoublePrecisionFloat 是从 Number 派生出来的），显然，不能使用嵌套类技术。

3.6 操作符 new 和 delete

C++为程序员提供了控制内存分配的方法。这种控制对“可调内存”这样的种类是重要的，它必须使类对象作为具体数据类型在给定的平台上运行。它也采用与语言类型系统集成的方式，使得控制逐类地分配内存成为可能。作为缺省，显而易见，程序大部分的实现最终还是要使用 malloc 和 free（C 内存分配原语）来管理用户访问的存储区。存在如下几个理由，要求部分或全部超越这些原语的使用：

- 内存可能需要静态设计并嵌入到系统中，诸如通信系统和飞行导航系统，而对象可能需要从固定池中分配。
- malloc 和 free 性能对用户的作用意图不够充分，特别是当存在着大量的小对象时，简明的算法能够比缺省的原语具有更好的性能。
- 现存的环境可能不提供内存管理子程序，所以程序自己必须提供内存管理子程序。

在分配存储区中的内存采用操作符 new 来操作，即

```
new int
```

为类型 int 的一个对象分配，或者

```
new int[10];
```

为类型 int 的 10 个对象的向量分配。当在类型引用 new 时，编译安排指向合适类型的返回值，在上述情况中，其类型是 int*。赋值语句为

```
int *ia=new int [size];
```

设定由 ia 指向的、分配用户所指定的尺寸的向量。创建动态地分配整数聚集（在这里未初始化）是作为所有内置类型的实例动态地分配。用 new 分配的类对象由构造函数初始化。

要回收对象向量的内存，我们可以写成：

```
delete [] ia;
```

其中取 ia 中的值，以前分配内存用 new，现在回收内存用 delete。当删去动态分配的对象向量时，使用[]，并且必须调用析构函数。一个单一对象，应按以下方式进行分配：

```
ia=new int;
```

回收单一对象时，用

```
delete ia;
```

下面我们来看整个内存管理的情况。我们既可以对快速内存管理算法进行组合，又可以在没有内存管理支撑的环境中运行 C++（例如，在微处理器外置独立程序）。这样，我们就能够用重定义 new 操作符和 delete 操作符的方法来获得其中任何一个。operator new 和 operator delete 定制的版本在例 3.12 中给出。这个分配符采用了一种简单的算法，此算法一直在异质程序块的单一表上使用，并且对满足内存要求进行合理的检索。一旦释放此表，程序块似乎用 struct Head 这样一个数据结构来维护释放表的链接来处理。

例 3.12：operator new 简明分配符。

```
// 从 stddef.h 中得到 size_t
```

```
#include < stddef.h>
struct Head {
    long length;
    struct Head *next;
}
static Head pool = { 0,0 };
static Head *h = (Head*)HEAP_BASE_ADDRESS;
/*快速简易找到第一个合适的分配器 */
typedef char *Char_p;
// 下一行依赖于机器和编译程序:
const long WRDSIZE = sizeof(void*);
void* operator new(size_t nbytes)
{
    /*首先，在空闲表中查找 */
    if (pool.next) {
        Head *prev = &pool;
        for (Head *cur = &pool; cur; cur = cur ->next) {
            if (cur ->length >= nbytes) {
                prev ->next = cur ->next;
                return (void*) (Char_p(cur) + sizeof(Head));
            } else prev = cur;
        }
    }
    /*在空闲表中没找到，于是就从堆中获得新块 */
    h = (Head*)(((int)((Char_p(h) + WRDSIZE - 1))/WRDSIZE) *WRDSIZE);
    h ->next = 0;
    h ->length = nbytes;
    h + = nbytes + sizeof(Head);
    return(void*) (Char_p(h) + sizeof(Head));
}
void operator delete(void *ptr)
{
    Head *p = (Head* )(Char_p(ptr) - sizeof(Head));
    p ->next = pool.next;
    pool.next = p;
}
```

现在无论什么时候 C++函数执行 new 或 delete 时都在标准库中用其中一个来调用重载操作符。采用重载操作符版本的内存管理算法可以实现用户所要做的任何事情。见例 3.12，从 HEAP_BASE_ADDRESS 开始使用堆存储块，一直无限扩充到内存的上限。

另一个通用的技巧是以一个类一个类为基础来分配定制的内存。这种方法有一个优点，即当运行在有它自己内存管理原语的操作系统时，因为用户可以提供一个算法，这样，采用特殊类的行为比通用目的操作系统的原语要灵巧。对经常分配小对象的类（通常是 4～10 个字节长度的小对象），当把它转换成使用这种方法时，可以极大地提高程序运行速度（用超过此数量级的缺省 malloc 的内存分配原语），也可以少浪费一些内存，避免了每分配一次 malloc 的 4～8 个字节的开销。

对任意给定的类来说，每个对象的内存分配总有相同的尺寸；对使用 new 来动态地分配类对象来说，我们可以预分配大量的内存，其尺寸是类对象长度的倍数。每个类可以有它自己的对象池，

以便精确地满足 operator new 对尺寸的要求。

例 3.13 说明了 String 具有它自己的分配操作符，这些操作符在全局层上重写了 new 和 delete 的语义。换言之，String 的公共界面与没有专门的存储管理是相同的。在 String 实现的联合体（union）中，允许既作为激活 String 对象来处理（这种情况下，联合体的内容把指针当做串来表示），又可以作为释放列表的存储块来处理（这种情况下，联合体的内容把链接当做释放列表）。

具有私有的、静态数据的 newlist，充当了释放列表的头。

例 3.13：具有定制存储管理的 String 的说明。

```
// 用哑串（不是一种具体数据类型）演示 new/delete
class String {
public:
        String() { rep = new char[1]; *rep='\0'; }
        String(const char *s) {
            rep = new char [::strlen(s) + 1];
            ::strcpy(rep, s);
        }
        ~String() { delete [] rep; }
    void*  operator new(size_t);
    void  operator delete(void*);
        ...   // 其他有趣的操作
private:
        static String *newlist;
        union {
            String *freepointer;
            char *rep;
        };
        int len;
};
```

例 3.14 给出了 String 类的操作符 new 和 delete，当 String 对象创建时，这些操作符重写了缺省的存储管理系统，表示为

```
new String;
```

当 String 对象分配内存，接着引用 String 的构造函数来初始化该对象。在构造函数里，串表示本身的内存作为字符向量的堆来分配，这与 String 的 operator new 互相不影响，但使用了系统缺省。这里，它的实现是维护内存块的链接表、每个 String 实例的大小，释放 String 对象就把该对象放在释放表最上头，从表上删除所分配的内存块。

例 3.14：String 类的操作符 new 和 delete。

```
String *String::newlist = 0;
void String::operator delete(void *p) {
    String *s = (String* ) p;
    s ->freepointer = newlist;
    newlist = s;
}
void* String::operator new(size_t size) {
    if (size != sizeof(String)) {
        // 当使用类派生（见第 4 章）和导出类没有 operator new 时，
```

```
        // 在这种情况下，需要做这种检查
        // 当它被回收时，String::operator delete 将回收
        // 越过尺寸块的域
        return malloc(size);
    } else if (!newlist) {
        newlist = (String*)new char [100 *sizeof (String)];
        for (int i = 0; i < 99; i ++ ) {
            newlist [i]. freepointer = &(newlist[i + 1]);
        }
        newlist[i]. freepointer = 0;
    }
    String *savenew = newlist;
    newlist = newlist ->freepointer;
    return savenew;
}
```

String 抽象隐藏了这些内存分配细节。在下面代码中，编译把 new String 的引用看做是要求从 String 的 operator new 分配内存（如果它出现），或者从::operator new（系统缺省）来分配内存（如果它不出现）：

```
int main() {
    String *p = new String("abcdef");
    delete p;
    return 0;
}
```

下面例 3.15 给出了带有静态内存池的 String 的类说明，例 3.16 给出了带有静态地址分配池的 String 的操作的实现。

所有类型的向量是用全局的 operator new 来分配的。虽然 String 有它自己的 operator new，引用

```
String *sp = new String [25];
```

其结果是按单一内存块来分配，长度为 25 * sizeof(String)字节，并且此块用:: operator new 来分配。这是因为该对象是按向量来分配的，向量是 C 的人为构件，而不是 C++类对象。

例 3.15：带有静态内存池的 String 的类说明。

```
class String {
private:
        static String *newlist;
        union {
            String *freepointer;
            char *rep;
        };
        int len;
public:
        enum { POOLSIZE = 1000 };
        String () { rep = new char[1]; *rep = '\0'; }
        String (const char *s) {
            rep = new char [::strlen(s) +1];
            :: strcpy (rep, s);
        }
```

```
        ~String () { delete[] rep; }
    void* operator new (size_ t);
    void operator delete (void*);
        ...      // 其他有趣的操作
};
```

例 3.16：带有静态地址分配池的 String 的操作。

```
String* String::newlist = 0;
static unsigned char memPool[String::POOLSIZE* sizeof (String)];
static const unsigned char *memPoolEnd =
    memPool + String::POOLSIZE + sizeof (String);
void String::operator delete (void *p) {
    String *s = (String*) p;
    s ->freepointer = newlist;
    newlist = s;
}
void* String::operator new (size_t size) {
    if (size != sizeof (String)) {
        // 当使用类派生（见第4章）和导出类没有 operator new 时，
        // 在这种情况下，需要做这种检查
        return::malloc(size);
    } else if (newlist == memPoolEnd) {
        StringOutOfMemory ();
        return 0
    } else if (!newlist) {
        newlist = (String* )memPool;
        for (int i =0; i < POOLSIZE - 1; i ++ ) {
            newlist [i]. freepointer = & (newlist[i + 1]);
        }
        newlist[i].freepointer = memPoolEnd;
    }
    String *savenew = newlist;
    newlist = newlist ->freepointer;
    return savenew;
}
```

3.7 把初始化从实例化中分离出来

实例化（instantiation）是对对象分配基本资源的处理，而初始化（initialization）是对配置对象状态的处理。有时，把实例化从初始化中分离出来是非常重要的。在独立的微处理器系统中，初始化或者重新初始化对象是重要的。当初始化相互引用对象时，也是有用的。

在第一种情况的一个例子中，我们考虑一个实时控制系统，它运行在独立的微处理器上。这个系统的初始化是由它的子系统的初始化来完成的，即用全局对象调用所属的构造函数来初始化。例如，从只读存储器（ROM）引用的全局表需要早期在具体的地址上初始化，所以在 ROM 中初始化代码能够被使用。对这些对象的内存大小和位置要预先确定，但其内容在运行时被初始化。如果在运行时系统发生故障，就可能要重新初始化被损坏的对象，而不必考虑其他对象。故障的外围设备

可以替换，CPU 无须停止运行（这是连续运行系统的一个重要考虑），并且可以引用在外围设备被替换后的外围（设备）对象的构造函数，其他对象照常引用在它的原始地址上的外围（设备）对象。

在第二种情况的一个例子中，我们考虑采用 Editor 和 Window 类的编辑器，其特点是每个成员指向另一个成员的实例，每个成员希望另一个成员为它自己初始化，所以存在着“环形”依赖关系。这可以采用分离构造函数和成员函数之间初始化的方法来加以解决。在两个对象已经分配内存并初始化之后，在成员函数中人工处理引用。但是，这与我们在原始位置上有构造函数的情形正好相反。我们要确保自动初始化，而无须程序员一定要记住所调用的初始化函数，并且它的结果仅存在于执行剩下的部分初始化对象的构造函数中。

“修补”的方法是不可取的，我们需要采用直截了当的程序设计的规定，迅速地表达创建和初始化的分离。本节所描述的不是“修补”的方法，而是首先用两个实例化来引导两个类的对象，然后再初始化两个对象。

这种方法是好的 C++程序设计实践的结晶。在 C++中，把初始化和实例化联系在一起，当对象存在时，消去了用户对显式引用初始化子程序的担心。另一方面，清除对象的析构函数也面临着相似的问题，所以这种方法应仅在环境要求时才可使用。这里所介绍的例子，将在第 9 章与更高级的习惯用法一起作进一步的讨论。

例 3.17：显式分配和回收对象。

```
#include < stddef.h >
void *operator new (size_t, void *p) { return p; }
struct S {
   s();
   ~s();
};
char buf [sizeof(S)] [100];
int bufIndex = 0;
void f() {
    S a;     // automatic allocation/initialization
    S *p = new S;                  // 显式分配
                                   // operator new 找到自动初始化存储区
    delete p;                      // 显式删除
                                   // 自动清除
    S *ppp = new (buf) S;          // 在 buf 中显式分配
                                   // 自动初始化
    ppp ->S:: ~S();                // 显式清除
    // 从 f()中返回“a”自动清除/解分配
}
```

我们来考察一下例 3.17 的代码。这里，我们对附加的参数重载了 operator new，即把该对象安放在内存地址位置上，operator new 的主体返回它的自变量。当附加参数提供给 new 时，这个 operator new 才在对象所处的地方直接调用，使用语法为

```
... new (object - address) Type (params...);
```

这样的引用妨碍了 new 自动分配内存的语义，并且在对象已知的位置减少了调用构造函数的次数。

我们也能够用类似的请求直接调用对象的析构函数：

```
object - address ->Type :: ~Type ();
```

代码可以指定对象的内存分配，任选的指针可以替代 buf。

注意，不能采用 delete 作为对象分配的方式。

例 3.18 介绍了另外一个例子，其中 foobar 是作为类型 Foo 的单个全局对象来加以使用。在编译时，地址空间对对象是不考虑的。这样就使代码可以运行在基本微处理器平台上，而此平台的操作系统是不支持动态内存分配的。或许会有某些计算要在 main 中来做，或者当 foobar 被执行初始化时，在另外一处产生的值是提供给 Foo 的构造函数的。在这个例子中，在不同的时间上，类 Foo 的不相关的两个对象可以重用相同的存储区（foobar）。

例 3.18：显式分配和回收到预分配内存中。

```
#include < iostream.h >
#include < stddef.h >
void *operator new (size_t, void *p) { return p; }
class Foo {
public:
    Foo() { rep=1; }
    Foo(int i) { rep = i; }
     ~Foo() { rep = 0; }
    void print() { cout << rep << "\n"; }
private:
    int rep;
};
struct { int:0; };   // 依赖于机器的调节
char foobar [sizeof (Foo) ];
Foo foefum;
int main ()
{
    foefum.print ();
    (&foefum) ->Foo :: ~Foo();              // 引起过早地清除全局范围的对象
    foefum.print ();                        // 未定义的结果
    Foo *fooptr = new (foobar) Foo;
    fooptr ->Foo :: ~Foo();
    fooptr = new(foobar)Foo(1);             // 与早期分配无关
    fooptr ->Foo :: ~Foo();                 // 引起过早的清除
                                            // 未做 fooptr 删除
    fooptr = new Foo;
    delete fooptr;
}
```

在例 3.18 中，过早地清除对象 foefum 在程序中不要做。如果做了，那么 foefum 的析构函数就被调用了两次!如果 foefum 重新实例化（new(&foefum)Foo），如果在程序出口处，该平台不去引用析构函数，或者如果该对象在内存中的分配不是原始对类对象的分配（如上面 fooptr）的话，则说明是安全的。

练 习

3.1 描述 struct，class，抽象数据类型和具体数据类型之间的区别。

3.2 如何理解有关在 C++语言中的列表（list），比较（compare）以及相对的抽象机制？

3.3 修改 String 类，以便子串可以从已存的串中提取，剩下的已存串原封不动。例如：

```
String s = "abcdefg";
...
String t = s(j, k);
```

在这个例子中，j 是起始索引，为 1，指示该串的第一个字符。如果 j 是负的，偏移量返回到串的结尾。为-1，表示是该串的最后一个字符。参数 k 表示许多字符从索引 j 开始。如果 k 是负数，那么 j 是作为子串最右边的索引，并且字符再返回到左边（该子串不是用这种“向后抓取”的移项）。

3.4 “X::X(const X &)”构造函数是规范格式的一部分。它的用途之一是在栈上提供拷贝函数调用参数的一种机制。在调用活动记录上，使得能拷贝调用的变量。给出具有 X::X(X)的构造函数（而未采用规范格式 X::X(const X &)）类 X 的对象将会发生什么事情。

3.5 编写 Node（结点）的一个构造函数，采用 String::String(Node)产生带有大括号的 String，并且从 String 中产生一棵完整的树。

3.6 只用静态设计内存来编写一个系统内存分配符，即在运行之前，用程序调整为固定尺寸和位置的内存。

3.7 如果所有类都对使用 operator->的 String（见例 3.11）遵循规范格式的话，那么 C++程序从未在应用层上使用任何指向类对象的指针（即类的外部使用它们自己的 operator->）。请对此加以说明。

3.8 考虑如下代码：

```
class String {
public:
   ...
   String (char *s) {...}
   operator const char* () const {...}
   ...
};
extern "C" {
   int open (char*, int ...);
}
int open (String, int ...);
```

下面的每种情况会发生什么呢？为什么？它们有相同的语义吗？为什么是或为什么不是（可以运行某些语句以便帮助理解此含义）？

```
int i = open ("abcd", O_RDONLY);
int j = open (String ("abcd"), O_ RDONLY);
String abcd ="abcd";
int k = open (abcd, O_RDONLY);
```

此练习可以帮助读者理解一些概念，以避免在读者自己的设计中一些模糊认识。通常，重载是作为系统设计问题来处理的，所以用户的说明与这些不相干。

3.9 对大小相当的C++程序，我们试图作一些变化，以使它的类的析构函数赋给this的值为零（如果读者所使用的编译器支持的话）。比较一下原始的C++程序与修改后的C++程序在性能方面有何不同。能进行相应解释吗？

3.10 描述实现strlen(String)函数（用第3.3节中所述方法）的另一种方法，替换使用转换成员函数的方法。

3.11 创建一个具有重载operator+的类Int，以使

```
Int i=0, j=0;
j = i+25+15+14+2;
```

其含义是020014004，同样地，要安排转换位off。这种形式读者能在什么地方使用？创建此重载符会有什么问题（此问题参见Lippman的一个例子[1]）？

3.12 从String类开始，添加重载符operator&()，产生一个StringPointer对象作为结果。StringPointer对象习惯于在程序中使用，通常在String*处使用。完成StringPointer的实现，包括operator->和operator*()。对String这样的改变是必要的。在什么地方可以使用这两个类？比较与之相反的在本章“计数指针”一节中所讨论的方法。

1 Lippman, Stanley B, A C++ Primer, Reading, Mass: Addison-Wesley, 1989.

第4章 继承性

继承是一种表达类与类之间特殊关系的语言特征，把类作为程序设计语言的抽象，按规范的格式进行处理，组织成抽象的层次结构。这些高层次的抽象构建了面向对象程序设计的基础。

继承通常扮演着两种角色。一种是抽象机制，即把类组织成有条理的特化层次结构的工具；另一种是作为可重用的机制，是创建与现存的类很相似的新类的一种方式。在这两种情况中，用户可以认为继承是把一个类二次转包给另一个类，或者继承是作为另外类的一些任务的代理。

继承最常用的方式是抽象机制。在C中，函数是算法的抽象机制，而struct是数据的基本抽象机制。一个类的C++概念是把这两种概念纳入到抽象数据类型和它的实现之中。在C++中，把这些抽象和强大的类型进行装扮，就像语言的内置类型一样提供给编译器，这是一种自身的强有力的抽象技术。继承进一步发展，提供了一种机制，即把相关类收集到一起，组成具有所有特征的高层概括。与类相似，相容的行为也可以组织成层次结构。顶层的类是所有下层的类的抽象，用屏障把程序员与类的实现细节隔离开，不受其变化的影响。

继承的含义就是从旧的类中构建新的抽象，C++像大多数面向对象语言一样，其中一个类是从另一个类中继承了数据和成员函数。但在C++中，继承所定义的类型也是兼容的，即两种类型可以紧密相关，一种类型的对象可以使用另一种类型中的一个对象。继承习惯用于表述这些关系，而编译类型检测允许用于可交换的兼容类。在第6章我们将要讨论：如果从类A导出类B，那么就隐含着类B的对象能够用在类A的对象中。在继承层次顶部的“父类”称为基类。在C++中，任何中间父类都称为该类的基类；有时，其他程序设计语言也称基类为超类。这个新类继承了父类的特性，被称为导出类；有时，其他程序设计语言也称导出类为子类。继承层次原则上可以有任意深度，但最好不要超过3～4层。

在继承层次结构中的类可以看做是基类抽象特征的不同形式，由基类编写的代码能够用于任何其他的可互换形式。多态性是处理类程序能力的一种强有力的抽象机制，它把程序员从导出类的实现细节中解脱出来。我们可以对相关的类使用继承关系，在应用中这些相关的类对应于激活的资源、产品、顾客等等。继承也能作为设计的辅助手段来帮助管理系统结构。

另外一种观点认为，继承作为已有类的增量式的改进来定义一个新类。如果需要一个新类，它的行为比已有的类多一些，那么只要稍微修改一下已有的类的实现，通过继承表达这种强有力的方式就可以达到目的。也就是说，新类可以继承已有类的所有特性，并可以在新类中用重写成员函数的方法来定制新的特性。

这样，继承可以使用以下两种不同的方式：

(1) 类与类之间的某些语义关系。这种关系是演化的结果，其中类的成员函数用于改善其特性，

或者特化已有基类的某些特性，而对其余的特性则加以继承。继承的这种使用可以获取一种在两个已有抽象行为之间更相似的能力，所以导出类能够用于基类的对象中。这种类型关系，常用以下两种方式来实现：

——公有继承；

——成员函数是虚拟的。

（2）代码共享关系。虽然一个类不可能展示它的基类的所有特性，但它可以通过继承充分地获得可重用，这种花费是值得的。虽然 Queue 和 CircularList 不可互换，但它们已有的公共代码，通过继承可以使这两种抽象分享已有公共代码。这种类型的关系常用以下两种方式来实现：

——私有继承；

——成员函数不是虚拟的。

注意，选择用 public 继承还是用 private 继承是实现时应当考虑的问题，更是设计时应当考虑的问题。它们分别将在第 5 章和第 6 章虚拟函数中加以讨论。在本章的例子中，PathName 强调了代码共享方法，而 Imaginary 类(虚数类)和 Telephone 例子塑造了类与类之间语义设计的关系。

在类的层次结构中，较低层次的类相对来讲表示特定的具体的概念，而较高层次的类相对来讲表示一般化的泛化的概念，分层概念隐含的关系就是继承。一个导出类可以继承父类的各种属性，同时又可将自己的或者继承来的属性传递到它的导出类中。利用继承性，可以在具有特殊性的已有类的基础上建立新类，而不必将所有的事情都从头开始设计，从而加快了对问题解的处理速度，降低了软件开发的复杂性和费用。

4.1 简单继承

在简单继承中，每个类可以有多个导出类，但最多只能有一个父类，从而形成一棵树结构。

我们在例 4.1 中将介绍具有继承性的复数类。复数类在科学研究和工程应用中非常有用，它是有理数（rational）、虚数（imaginary）、实数（real）和整数（integer）的概括。有理数、虚数等的代码，对复数（Complex）来说，可以采用增量修改代码的方式来构造。这些类是继承得比较完美的例子。

例 4.1：一个简单的复数类。

```
class Complex {
public:
    Complex(double r = 0, double i = 0): rpart (r), ipart(i) {}
    Complex(const Complex &c): rpart (c.rpart), ipart (c.ipart) {}
    Complex & operator = (const Complex &c) {
        rpart = c.rpart; ipart = c.ipart; return *this;
    }
    Complex operator+(const Complex &c) const {
        return Complex(rpart + c.rpart, ipart + c.ipart);
    }
    friend Complex operator+(double d, const Complex &c) {
        return c + Complex(d);
    }
    friend Complex operator+(int i, const Complex &c) {
        return c + Complex(i);
```

```
    }
    Complex operator-(const Complex &c1) const {
        return Complex(rpart - c1.rpart, ipart - c1.ipart);
    }
    friend Complex operator-(double d, const Complex &c) {
        return -c +Complex(d);
    }
    friend Complex operator-(int i, Const Complex &c) {
        return -c +Complex(i);
    }
    Complex operator* (const Complex &c) const {
        return Complex (rpart *c.rpart - ipart *c.ipart,
                rpart* c.ipart+c.rpart *ipart);
    }
    friend Complex operator* (double d, const Complex& c) {
        return c *Complex(d);
    }
    friend Complex operator* (int i, const Complex& c) {
        return c *Complex(i);
    }
    Complex operator/(const Complex &c) const { ... }
    ...
    operator double() {
        return ::sqrt(rpart* rpart + ipart* ipart);
    }
    Complex operator-() const { return Complex (-rpart, -ipart); }
private:
    double rpart, ipart;
};
```

下面是从 Complex 类导出的简单虚数（imaginary number）类：

```
class Complex {
friend class Imaginary;
public:
    ...
};
class Imaginary:public Complex {
public:
    Imaginary(double i = 0): Complex (0,i) {}
    Imaginary(const Complex &c): Complex (0,c.ipart) {}
    Imaginary& operator = (const Complex &c) {
        rpart = 0; ipart = c.ipart; return *this;
    }
};
```

注意，我们所构造的 Imaginary 类是 Complex 类的一个友元。在这个继承的层次结构中，这些类被当做软件包而一起编写的，其效率也是与这些类紧密相关的。作为一般规则，这些类应当避免进入它们的内部结构，对导出类也不例外（因为继承不是违背封装原则的脱词）。另一方面，这个例子说明了如何用父类导出子类以及如何设计一个好的基类。

类的层次结构所依赖的交义层在这个例子中变得很清晰。考虑下面这一段应用代码：

```
int main() {
    Complex a (42,2);
    Imaginary b = 2;
    Complex c = a-b;
    // c是Complex(42,0)
    return 0;
}
```

这里，用 Complex::operator-返回结果，它是作为 Complex 的静态类型，但返回的值实际上是实数。这就产生了两个问题：

（1）在 C++的特有的算术类型中，以及在内置类型（如 double）和用户定义类（如 Complex 和 Imaginary）之间产生了二义性。我们希望能像 double 类型那样固定在 Complex 层次结构的某个位置，而且特别是对从 Complex 导出的 double（毕竟 double 刚好是 Complex 的一个类型），更要固定在 Complex 层次结构的某个位置。但 double 类型是一个内置类型而不是一个类，因此我们不能把这种构造纳入继承的层次结构中。这样告诉编译器，double 和 Complex 为兼容的类型是很不合适的。我们最好能对算术类型作这样的安排：所有 double 算术类型都转换成 Complex。这种选择是对每个使用完整的类的 double 的替换：

```
class Double:public Complex {
    // Double是一种Complex
public:
    Double(double r = 0): Complex(r) {}
    Double(const Complex &c): Complex(c.rpart) {}
    Double& operator = (const Complex &c) {
        rpart = c.rpart; ipart = 0; return *this;
    }
    ...    // 对Double进行操作优化
};
```

这仅解决了第一个问题。

（2）更一般性的：这里，我们有一个 Complex 成员函数，提供了一个 Imaginary 参数，期望产生一个 double 的结果。一般地，在这个继承层次的大多数类中需要知道其他类的更多的信息，这个解完全与面向对象程序设计有关。

为 Imaginary 语义裁剪 Complex 操作

Imaginary 类从它的父类（Complex 类）继承了大部分操作。但我们要确保 Imaginary 对象总有一个零的实部，而 Imaginary 类重新定义这些操作可能在其他方面变成了无效的假设。例如，Complex 对 Imaginary 的赋值需要作特殊处理，使 Imaginary 最终没有一个实部。我们既可以处理运行时的错误问题，又可以处理当执行赋值时实部部分为零的操作。这里，我们采用后者。转换会丢失信息（实部被丢弃），而这与把 double 强制转换成 integer 时要丢失小数部分的信息是一致的。

仅能给出 rpart 域为非零值的其他基类成员函数是 Complex::Complex(double,double)，但它不能被 Imaginary 继承。基类的构造函数不能被导出类继承。

在导出类中重用基类代码

继承性使 Imaginary 重用了 Complex 算术操作成员函数的代码。我们已经建立了新的类型，它

通过已有类型的成员函数来完成大部分工作。因为虚数也是复数，所以这样做是安全的。虚数是一种特殊的复数，例如，我们把 Imaginary 添加到 Complex 中就会产生一个新的 Complex：

```
int main() {
    Complex c = Complex(1,2);
    Imaginary k = -1;
    Complex sum=c +k; //合法: c.operator+ (const Complex&)
    // sum 是 Complex(1,1)
    return 0;
}
```

虽然 Complex::operator+用一个 Complex 作为参数，Imaginary 是 Complex 的一种类型，因为在类之间有继承关系，所以编译器知道这个事实。Complex 在接口中提供的任何信息都被 Imaginary 的接口所继承。C++的公有继承（public inheritance）说明获取了那种关系，允许对象说明为 Imaginary 类型，这正是 Complex 类所期望的。这样，继承定义了在相似类之间的兼容类型。

调整导出类成员函数效率

当对 Imaginary 进行应用时，Complex 成员函数工作得很好，但有些工作是不必要的。例如，引用 operator double，返回作为 double 的复数量值，数的两部分倍数并返回实部加虚部之和的平方根。对 Imaginary 来说，这些是算法的工作，但该算法可以简化。如果已知它的实部总是为零，则可以提高 Imaginary 的执行效率。我们可以对 Imaginary 类增加一个 operator double 版本：

```
Imaginary::operator double() {
   return ipart;
}
```

这个新版本将作为 Imaginary 的变量说明而加以引用，而老版本将仍旧被 Complex 所使用：

```
int main() {
    Imaginary i = -1;
    Complex j = Complex (0, -1);
    double a,b;
    a = double (i);        // Imaginary::operator double
    b = double (j);        // Complex::operator double
    // 都产生 1
    ...
}
```

从该例子中我们可以看到，对每个基类描绘了大部分的导出类情形。当然，复合类可以由单一基类派生出来。在例 4.2 中，我们共享公共基类的电话（telephone）系统的几个类型的共同属性。注意，Telephone 类没有产生对象，在现实生活中抽象电话是不存在的。Telephone 类仅作为一个框架而存在，或者作为具体的电话种类的模板而存在。下面，我们检查某些基本机制，以控制在基本机制中 Telephone 实例化的对象。但是，Telephone 类具有如何打电话的一般知识。例如，它知道如何寻找捕获终端变量或打电话的上下文信息，或者其他系统信息，从而决定与打电话有关的状态。此外，我们还假定可以用与电话系统无关的方式来执行，即对 POTSPhone 使用相同的机制（简单的老式 Telephone 服务的电话系统），作为 ISDNPhone（集成服务的数字网络电话，具有先进特征的数字电话）。这就意味着我们可以写出下面的编码：

```
POTSPhone phone1, phone2;
...
if (phone1.isTalking())phone2.ring();
```

POTSPhone 类继承了 Telephone 类的 ring 和 isTalking 操作，就好像这些操作是已经被物理地复制到该导出类中一样。这就实现了无须代码复制的代码重用。

例 4.2：一个公共基类派生出两个导出类。

```
class Telephone {
public:
   void ring();
   Bool isOnHook();
   Bool isTalking();

   Bool isDialing();
   DigitString collectDigits();
   LineNumber extension();
   ~Telephone();
protected:
   LineNumber extensionData;
   Telephone();
};
// POTS 是普通的电话服务系统
class POTSPhone:public Telephone {
public:
   Bool runDiagnostics();
   POTSPhone();
   POTSPhone(POTSPhone&);
   ~POTSPhone();
private:
   // 这些是在电话办公系统中电话线的连接细节
   Frame frameNumberVal;
   Rack rackNumberVal;
   Pair pairVal;
};
// ISDN 是集成服务数字网络电话系统
class ISDNPhone:public Telephone {
public:
   ISDNPhone();
   ISDNPhone(ISDNPhone&);
   ~ISDNPhone();
   void sendBPacket();
   void sendDPacket();
private:
   Channel b1, b2, d;
};
class PrincessPhone: public POTSPhone {
   ...
};
```

4.2 作用域和访问控制

我们已经研究了如何利用继承，结合已有的类来创建新类，从而使我们的设计不必将所有的事情都从头开始。如果我们关注封装和信息隐藏，那么就必须仔细研究继承如何影响类成员的访问，类与类之间都与继承有关，而且一个类可以对另一个类提供一些服务。这里与继承有关的类与类之间的成员的访问表示为“纵向访问”。在继承的层次结构中，基于类的观点的相互关系是“上”或者“下”。类成员对现实世界外部的访问，我们称为“横向访问”，类与类之间的关系与程序结构处于同等地位。继承所产生的成员访问控制问题取决于继承的性质：公有、保护和私有。

“纵向”访问控制和继承

一个类可以控制由导出类构造的成员，而导出类不能访问基类的 private 成员。我们来看下面的代码：

```
class A {
public:
    A();
    ~A();
    ...
private:
    int val;
};
class B: public A {
public:
    B();
    ~B();
    ...
    void func();     // 不能访问 A::val
};
```

B 没有成员函数可以访问 val，即使 val 实例是出现在每个 B 对象中。这里有一堵“防火墙”，沿着纵向（继承）作用域，保护着从导出类产生的基类成员的 private 访问。导出类并没有比其他类有更多的权限违背基类的封装性。

如果试图造就这样一种私有成员，它既能访问类 A，又能访问类 B，这是绝对办不到的。如果 val 是 public，那它就能够访问 B 的（或者其他任何一个的）成员函数。如果我们要使一个或多个 A 的成员访问 B，而不说明它们为 public，那我们就需使用 protected：

```
class A {
public:
    A();
    ~A();
    ...
protected:
    int val;
};
class B: public A {
public:
    B();
```

```
    ~B();
    ...
    void func ();      // 能够访问 A::val
};
class C: public B {
public:
    c();
    ~c();
    ...
    void func2 ();      // 能够访问 A::val
};
```

这里，除了它的成员或它的导出类的友元函数之外，val 的行为仍然是 private 的。

为了限制对 val 的访问，即仅对导出类 B 而不是 C 的访问是合法的，我们可以使用友元来构造：

```
class A {
friend class B;
public:
    ...
private:                        // 全部用墙隔离，除非 A 的成员是友元(friend)
     int val;
};
class B: public A {
public:
    ...
    void func();                // 能够访问 A::val
};
class C: public A {
public:
    ...
    void func3();               // 不能够访问 A::val
};
```

这里，导出类 B 对 A 有一种特殊的关系，即允许访问这个导出类的基类的 private 域。类 C 对这样的访问被排除在外。构造一个基类成员 protected 类似于构造所有导出类的基类 friend。如果所有类已经访问它们的基类的 protected 成员，故基类不需要每时每刻修改一个新的导出类（要求访问这样的成员）。如果带有 friend 的话，则对每个新的导出类的添加，都要求在基类中有一个新的 friend 子句。另一方面，protected 子句一味地允许访问所有的导出类，而友元关系却可以在一个类一个类的基础上获得访问。一个 protected 成员能访问这个导出类的友元或者成员函数，但这仅仅是通过指针、引用或对该导出类的对象进行访问。这就是为什么在例 4.1 中 Imaginary 要构造成 Complex 的一个友元的原因，它替代了使用 protected 子句。特别要指出的是，导出类赋予的操作必须访问一个基类的成员对象（作为一个参数的情形）：

```
Imaginary &Imaginary::operator = (const Complex &c) {
    rpart = 0;
    ipart = c.ipart;
    return *this;
}
```

如果基类成员 rpart 和 ipart 是 protected，则编译时将产生错误的结果。

作为添加的一个例子，让我们来考虑建立一个 PathName 类（由第 3 章的 String 类得到），重写为

```
class String {
public:
   String()                          {rep = new StringRep(" ");}
   String(const String &s)           {rep = s.rep;rep ->count ++ ;}
   String(const char *s)             {rep = new StringRep(s);}
   ~String()      {if (--rep ->count <=0) delete rep;}
   String &operator = (const String &s) {
      s.rep -> count ++;
      if (--rep -> count <=0) delete rep;
      rep = s.rep;
      return *this;
   }
   String operator+(const String&)const;   // 在另外一个地方
   operator const char* () const {return rep -> rep;}
   String operator() (int, int) const;
            // 子串
   int length() const                {return::strlen(rep -> rep);}
protected:
   StringRep *rep;
};
```

PathName 可继承 String 的所有特性：

```
Class PathName: public String {
public:
   PathName(const String&);
   PathName() : baseNameRep(" "), String () {}
   PathName(const PathName &p) : String(p),
      baseNameRep(p.baseNameRep) {/*空的*/}
   PathName &operator = (const PathName &p) {
      String::operator = (p);    // 指定基类部分
      baseNameRep = p.baseNameRep;
      return *this;
   }
   ~PathName()          {/*空的* /}
   String baseName()      {return baseNameRep;}
   String prefix();
   String suffix();
   String fullPathName()   {return *this;}
   String dirName() {
      return(*this) (0, length() -baseName(). length());
   }
   String changeBaseName(const String &newFile);
private:
   String baseNameRep;
};
```

因为 PathName 明显地由 String 派生出来，所以在 PathName 对象上，用户可以引用任何 public String 成员函数。我们要保护在 PathName 上用户执行 substring（子串）的操作。这便维护了用户可

以遵循路径分量进行修改，除非它通过其他路径。操作在路径上的高层成员函数，则根据它们的路径分量目录进行，而不是由字符串来替代（例如，changeBaseName）。另一种方式是在运行时设置访问陷阱并打印信息：

```
class PathName: public String {
public:
    ...
    String operator() (int, int) {
        printf("illegal PathName substring\n");
        return String(" ");
    }
    ...
};
```

但是，这个无效引用直到运行时才被发现，这太晚了。我们宁愿在编译时发现有错。如果PathName 类重新定义子串(substring)操作使之不能访问的保护部分，则编译时就能保证不对PathName 对象强制引用 operator()(int,int)。这也就是说，虽然 operator()(int,int)在 String 上是有效操作符，但对 PathName 类来说却是无效的。即意味着 PathName 类没有继承 String 类的特性，或者至少是没有全部继承。

如果我们要精确地限制使用 PathName 类的对象访问 String 类的接口的那一部分代码，那么必须使用私有继承。下面的代码告诉我们：作为 PathName 类的接口一部分仅禁止 const char*转换成String 类的操作符操作：

```
class PathName: private String {
public:
    String::operator const Char*;    //从基类
    ...      // 其他 PathName 成员函数
};
```

这称之为“访问标识符”，它不改变操作符的实现或建立一个新的操作符。它是刚刚建立的PathName 的 public 接口操作符的一部分，替代了用私有导出被隐藏的那部分。其他操作符和成员函数，例如，operator()(int,int)，剩下来的 private 到 String 部分在 PathName 接口中没有说明。如果下面代码出现在 String 或 PathName 外部：

```
PathName p;
...p(1,3)...
```

因操作符是私有的，因此编译会产生一个致命的错误。但是，基类 String::operator()(int,int)仍能访问任何 PathName 成员函数。

“横向”访问控制和继承

上述构造是从基类到导出类习惯采用的访问控制。这些相同的构造，习惯用于控制什么样的基类操作对类的用户是可行的。

当一个基类对一个导出类访问它的成员有一些控制过载问题时，该导出类对该导出类的客户访问基类接口部分也有一些控制过载问题。这里存在着两种支持机制：一种是使用不同的继承模式，告知用户如何将一个导出类宣布为基类成员；另一种是在一个成员一个成员的基础上使用访问标识符。

对继承成员大多数共同的构造是习惯使用控制访问是公有派生（public derivation）的，也称为

公有继承（public inheritance）。对一个基类 A 和一个导出类 B 来说：

```
class A{      // 基类(超类)
public:
    ...
};
class B:public A {
public:
    ...
};
```

所有 A 的 public 成员对 B 的成员是可用的。如果要替代：

```
class A{      // 基类（超类）
public:
    ...
};
class B:private A{
public:
    ...
};
```

对类 B 的用户来说，A 没有一个成员对 B 的成员访问是有效的。这就称为私有继承（private inheritance）。但是，类 B 的成员函数仍可访问类 A 的 public 和 protected 成员函数。

如果一个类从某些基类私有地派生出来，我们可以在它的 public 接口上挑选基类 public 接口部分进行访问。为了达到这个要求，使用第二种方法控制访问继承成员，即采用访问标识符。我们在上面已经构造了 String 类和 PathName 类。下面我们来考虑这样的类：

```
class List {
public:
    virtual void *head();
    virtual void *tail();
    virtual int count();
    virtual long has (void*);
    virtual void insert (void*);
};
```

而从这个类的私有继承导出下面另一个新类：

```
class Set:private List {
public:
    void insert (void *m);
    List::count;    // 访问
    List::has;      // 标识符
};
```

这里，Set 用于 List 的功能上的实现。因为它们没有作为集合操作的概念，所以对它的基类（head 和 tail）所隐含的 public 成员函数使用私有继承。继承就是这个含义，除了对 insert 操作，因为 List 的语义与 Set 的语义不同（列表可以多次复制一个元素，集合则不能）。

注意，不能逆向操作。例如，下面的做法是非法的：

```
class Set:public List {
```

```
private:
   List::head;    // 非法
   List::tail;    // 非法
public:
   void insert (void *m);
};
```

对上面的代码编译器会产生一个错误信息。从 List 公共继承的 Set，断言 Set 的接口所做的任何事情就是 List 接口部分所做的事情。这就意味着 List 对象能够去做任何一件事，而 Set 的对象应该能做得更好。C++类型系统具有这个优点。如果使用公共继承接收一个 Set 对象，可以在任何地方接收所期待的一个 List 对象。但对私有继承来说，会产生如下异常情况：

```
void *listhead (List l) {
   return l.head();
}
int main(){
   Set s;
   void *p = listhead(s);
   return 0;
}
```

试问：引用 l.head()是有效还是无效？如果编译允许的话，那对程序员来说真是不可想像，即指定的 head 应该对 Set 对象来说是 private 的，这就构成了对封装性的违背。因为 listhead 可以从 main 开始分开编译。编译器在上述例子中不可能提供编译时间上的诊断。作为对这个情况的保护措施，编译不允许这种可能的存在，因而成为非法。也就是说，对基类比对导出类的访问需要施加更多的限制。

公有继承几乎应用于所有的情况，而私有继承通常用于导出类对基类代码的可重用方面。大多数时候，其他的可重用构造均优于私有继承（见第 6 章）。不幸的是，私有继承是缺省的（如果两者的关键字均未指定），所以，应当有意识地、小心地去构造大多数导出的公有类。

注意，一个基类不存在一种方式去强制它的导出类成员可以被访问的操作。这就是对该导出类的控制。

导出类可以定义一个具有同名的成员函数，作为在它的基类上已经存在的一个函数。从该类的外部来看，这个新的成员函数隐藏了基类的定义，即使该函数在参数的个数、类型以及返回值方面均有所不同。在例 4.3 中可以发现这种情况。在类 B 本身的成员函数，仅涉及到 A 的成员函数 f，是全部类作用域的解，即 A::f。

例 4.3：同名但不同署名的继承函数。

```
class A {                        // 基类(超类)
public:
   void f(int);
   void g(Void*);
   ...
};
class B:public A {               // 导出类(子类)
public:
   void f(double);
```

```
    void g(int);
    ...
};
void B::g(int k) {
    f(k);                       // 带提示的 B::f(double)
    A::f(k);                    // A::f(int);
    thiS ->A::f(k);             // A::f(int);
    ::f(k);                     // 非法：没有全局的 f
}
int main() {
    A *a;
    B *b;
    int I;
    ...
    a ->f(1);                   // A::f(int)
    a ->f(3.0);                 // 带强制 int(3.0)的 A::f(int)
    b ->f(2);                   // 带提示 double(2)的 B::f(double)
    b ->f(2.0);                 // B::f(double)
    a ->g(0);                   // A::g(void*)
    b ->g(0);                   // B::g(int)
    b ->g(&i);                  // 错误：B::g 隐藏了 A::g
}
```

如果一些代码使用基类指针或引用来访问类对象，那它仅能通过指针来引用基类成员函数。但是，如果一个成员函数使用点标记（如 object.function）直接引用导出类的一个实例，则导出类成员函数就选中了同名的基类成员函数。

这个类的用户可以显式地在特定层上挑选出超越作用域和引用的成员函数。看一下例 4.4，我们用添加一个全局的 f()稍微修改一下以前的例子。该全局函数在遵从 B::f(double)中（隐藏了全局变量），它忽略了与调用 f(6)的参数精确的匹配。在调用::f(5)中，全局函数 f 用全局作用域操作符(::)进行显式引用。

例 4.4：在不同作用域中同名的函数访问。

```
Void f(int j) {...}
class A {                       // 基类(超类)
public:
    void f(int);
    ...
};
class B: public A {             // 导出类(子类)
public:
    Void f(double);
    Void g(int);
    ...
};
void B::g(int k) {
    f(k);                       // 带有提示的 B::f(double)
    A::f(k);                    // A::f(int);
```

```
    this ->A::f(k);            // A::f(int);
    ::f(k);                    // ::f(int);
}
int main() {
    A *a;
    B *b;
    int I;
    ...
    a ->f(1);                  // A::f(int)
    a ->f(3.0);                // 带强制 int(3.0)的 A::f(int)
    b ->f(2);                  // 带提示 double(2)的 B::f(double)
    b ->f(2.0);                // B::f(double)
    b ->A::f(7.0);             // 带强制 int(7.0)的 A::f(int)
    b ->g(10);                 // B::g(int)
}
```

具有继承和访问控制的控制实例化

有时，一个基类仅作为在建立导出类上的一个框架。例如，上述的 Telephone 类是服务于具有特殊行为的真实的数字电话类型：ISDNPhone 用于数字电话类型，而 POTSPhone 用于“普通老式电话服务”的电话类型。Telephone 类说明了所有这种电话的公共行为，以作为它们的成员函数的语义定义。它总是可以对所有与电话类型无关的（如，电话号码）公共的具体实现进行服务。

用 C++的保护构造，一个类对导出类提供了作为框架的接口，而且保护了本身建立的实例。我们来看下面 Telephone 的例子：

```
class Telephone {
public:
    ~Telephone();
    void ring();
    ...
protected:
    Telephone();
private:
    ...
};
class ISDNPhone: public Telephone {
    ...
};
class POTSPhone: public Telephone {
    ...
};
```

注意，Telephone::Telephone 是被保护的。除了用导出类构造以外，这个建议是它的构造函数不能被“调用”。如此说明的语义是，Telephone 类的一个对象不能被说明。一个抽象的电话不能存在于它自己的电话类上：它仅仅是标识符，真实的电话，像 ISDNPhone 和 POTSPhone 可以被实在地加以说明和实例化。所有“一般”或“抽象”电话的公共特性都是“特定”电话特征的一部分。当导出类被实例化时，基类的构造函数被隐式地调用。但该基类的一个对象不能在自己的基类上进行实例化。

事实上，这里，保护类型 Telephone 的一个对象不是最好的方式：一种是能在 Telephone 的任何

子类的成员函数被创建；另一种是如果这样的情况存在的话，Telephone 类的友元函数也能被创建。

在 5.4 节中将描述面向对象程序设计更强有力的语言构造，即更直接更精确地涉及到抽象基类的通信设计论题。

4.3 构造函数和析构函数

构造函数和析构函数是特殊的成员函数。它可以对新的类对象自动地进行初始化，并且在对象生命期结束时清除该对象。继承类的一个对象包含了几个不同的类的数据成员，并且所有这些类的构造函数和析构函数，对初始化和清除对象都产生了一定的影响。本节研究在继承的上下文中的构造函数和析构函数：它们的执行次序以及对参数化基类构造函数的一个类的机制。

构造函数和析构函数的执行次序

当使用继承时，由它自己类的成员函数来操纵一个对象，而所有的父类在该类之上。每个对象都有多个构造函数和析构函数，初始化和清除逻辑的手段是由单一成员函数集中提供的。

在继承层次结构中的类构造函数遵循预先执行的模式。编译自动地执行基类构造函数，并且把对象的构造函数置入到对象建立中。构造函数的主体代码仅做了一半的工作：编译也提供它自己的分配内存代码，并引用合适的、相关的构造函数。

当对象用 new 保留字创建时，也就动态地分配了内存。如果一个类存在的话，则既可以引用类自己的 operator new 函数，也可以引用全局的::operator new。如果该对象不是使用 new 动态地分配内存，那么构造函数就采用所提供的自动的和全局的变量来分配内存，或者用某种方式（例如，用导出类的构造函数）来预先建立。

如果构造函数属于导出类，它的首要工作就是调用它的基类构造函数。构造函数自动地完成这项工作，不需要程序员进行编码，但程序员能够对调用构造函数做一些控制。内存已经分配的构造函数依次调用它的基类构造函数。

其次，构造函数初始化任何它自己的数据成员，这个数据成员是引用它们的构造函数的类对象。这些对象通常是用它们的缺省构造函数来初始化的，但也能够通过初始化表来加以改变。

在构造函数已经决定用它的基类和成员对象初始化之后，就执行用户提供的构造函数主体。该主体可以对以前的初始化域指定新的值，该主体也可以初始化所有数据成员，它们是诸如 int，short，long，char 和 double 以及任何指针的内置类型。

当一个构造函数完成时，它就返回到调用的那个控制点。当一个基类构造函数完成时，它就返回到引用导出类构造函数控制点，并且这个导出类对象的初始化确保此控制点。最后一项工作执行的是在继承层次结构叶子上（大多数是导出类）的类中的初始化代码。

当编译构造函数时，通常编译添加隐含代码，以处理 C++内存管理、成员函数调度、基类构造函数调用和类对象它们自己的数据成员的初始化。正常情况下，用户可以忽略这些代码的存在。因为安装这样的代码依赖于版本，因此无论怎样都不应依赖于这样的细节。

析构函数执行的次序与构造函数执行的次序相反：首先，执行成员对象的析构函数，其次执行对象类的析构函数，然后再执行基类的析构函数。如果该类的操作数（这个类是在继承层次结构的叶子上）有（或者继承了）operator delete，那么析构函数调用回收内存以替代系统缺省调用。在析构函数返回之前，这个操作数对象本身释放内存是要做的最后一件事情。

向基类和成员的构造函数传递参数

让我们来看一看构造函数 Imaginary::Imaginary(double)的语法。注意，这个构造函数的主体是空的，它在执行时会产生副作用。如果用户已经对这个构造函数编写了简单的定义，

```
Imaginary::Imaginary(double d) {
   ipart = d;
}
```

那么编译将安排缺省的基类构造函数——Complex::Complex()——在做赋值之前，调用 Imaginary::Imaginary 的头。这样可以确保 Imaginary 完成 Complex 类（复数类）的初始化工作，并且 Complex 的数据能够用于构建 Imaginary 类。

Imaginary 构造函数的有效参数潜在地使用它的基类构造函数，并且程序员具有直接对调用具体的基类构造函数进行编译的灵活性。在前面的复数类的例子中，程序员清楚地指定在执行 Imaginary::Imaginary(double)之前引用 Complex(0，i)。

这里也应该注意，例 4.1 的 Complex::Complex(double，double)构造函数是如何指定对 rpart 和 ipart 的初始化的。下面两种行为视为等同的：

```
Imaginary :: Imaginary (double a,
double b)
{
   rpart = a;
   ipart = b
}
```

```
Imaginary :: Imaginary (double a,
double b)
   : rpart(a), ipart(b)
{
   /*空的*/
}
```

如果构造函数像上述右边那样直接被调用，与左边的引用赋值比较，则可能减少冗余的拷贝。PathName 的例子非常清楚地表示：拷贝到新对象中的元素是字符串而不仅仅是机器字：

```
PathName::PathName()
{
   rep = String(" ");
   baseNameRep = String(" ");
}
```

```
PathName::PathName()
   : baseNameRep(" "), String()
      {/*空的*/}
```

同样，baseNameRep 必须在它分配之前被初始化。在上述左边的例子中，在执行 PathName 构造函数语句之前，初始化基类 String 之后，引用 baseNameRep 上的构造函数 String::String()。注意，这意味着 rep 进行了双重初始化——一次是在基类构造函数的隐含调用的时候，另一次是在对 PathName 的构造函数进行显式调用时。

在构造函数 PathName(const PathName&)中，我们看到一个类似的构造。基类构造函数 String::String(const String&)是显式地把 P 作为参数来直接调用的。它在执行 PathName 构造函数代码之前被引用。

另一个构造函数的例子，为它的基类构造函数提供引用参数，其说明可见附录 B 的 Shapes 例子中的 Square 类。正方形(Square)是矩形的特殊情形，两者的区别仅用一种方法，即 Square 仅需要一个参数，而不是两个参数，描述它的边长即可。这比用 Rectangle 构造函数来处理要简单：

```
class Square:public Rect {
public:
```

```
    Square(Coordinate ctr, long x): Rect (ctr, x, x){}
};
```

这个新的抽象具有所有 Rectangle 的能力和行为，它的创建只需要 4 行代码。

4.4 类指针转换

利用继承的特性，我们可以使设计简化，并易于对大系统进行维护。继承最强大的能力是在继承的层次结构（就像加入兼容的抽象一样）中支持用通用应用代码处理对象类。所有这样的对象可以被处理成好像是在树根上的类的实例一样，这就是多态性。对象的许多不同的形式能够用同样的方式来进行处理，这就意味着应用代码对 Telephone 具体类型的实现细节可以忽略（如对 POTSPhone 和 ISDNPhone），并只处理所有像 Telephone 那样的类型。

这种程序设计能力基于指向与继承有关的类对象的指针之间的转换。我们发现，大多数 C++程序都用对象指针进行存取，该指针经常是作为对象从堆中进行分配的。C++指针变量提供了两种在一般实例变量中找不到的灵活的特性。第一种特性是基类的指针可以指向在隶属于导出层次结构中任意一类的一个对象；第二种特性是成员函数通过指针来引用，这个指针是在运行时选取的，而不是在编译时确定的。当然，此成员函数要说明为 virtual。这两种特性是面向对象程序设计中最基本的。如果一个类引用变量被捆绑在一个对象上，那么成员函数就通过这个引用变量来调用，同时，也接受多态性处理。但是，更普遍的情况还是采用指针。而对一个对象使用“.”点操作符（如 officePhone.ring）的成员函数的直接应用，并不提供这种灵活性。

通常情况下，如果说明一个指针指向类 C 的一些对象，那就可以使用指向任何一个子孙类的对象的地址指针，或者指向另外的第 *n* 个产生的导出类。但不能总是反其道而行之。所以，例子中 Telephone*可以习惯用于指向任意一个 Telephone 对象、ISDNPhone 对象或 POTSPhone 对象。有人认为，Telephone 可以指向 POTSPhone，这是因为前者的特征是用后者“覆盖着”。与之相反的做法则是非法的，因为不是所有的 POTSPhone 的行为都可以用 Telephone 所覆盖，这对所有的电话来说并不是公有的特性。这类错误在编译时可以检测到。

我们可以在显式的状态下，把基类的指针转换成导出类的指针。如果基类成员函数应用等价于所有导出类的对象，那么一般的基类指针可以用于层次结构中任意一个对象的成员函数，无须一定要准确知道该对象的类型。这里特别应用的另一个例子是电话类层次结构的外部函数，即对电话对象应用一般的电话类操作，无须一定要确切地知道在操纵时电话是什么样的类型。这里，ringPhones 函数用指向 Telephone 的异质向量指针作为参数：

```
void ringPhones(Telephone phoneArray[])
{
    for (Telephone *p = phoneArray; p; p++) {
        p ->ring();
    }
}
int main()
{
     Telephone *phoneArray[10];
     int i=0;
```

```
    phoneArray[i++] = new POTSPhone("5384");
    phoneArray[i++] = new ISDNPhone("5010");
    phoneArray[i++] = new ISDNPhone("0");
    phoneArray[i++] = new POTSPhone("5383");
    phoneArray[i++] = 0;
    ringPhones (phoneArray);
    ...
}
```

所要做的事情是 Telephone::ring 被每个放入 phoneArray 的电话对象引用。但这种方法不能获得任何数据，或者不能获取导出类定制的成员函数，因为 ringPhones 仅在编译时才能够知道基类 Telephone。例如，让我们在下面的 OperatorPhone 类运行它自己定制的 ring 操作，然后运行并稍微修改一下主程序 main()：

```
class OperatorPhone: public ISDNPhone {
public:
   OperatorPhone();
   OperatorPhone(OperatorPhone&);
   ~OperatorPhone();
   void ring();    // 专门的响铃所需要的操作符
};
int main()
{
   Telephone *phoneArray[10];
   int i = 0;
   phoneArray[i++] = new POTSPhone("5384");
   phoneArray[i++] = new ISDNPhone("5010");
   phoneArray[i++] = new OperatorPhone("0");        // 有它自己的 ring
   phoneArray[i++] = new POTSPhone("5383");
   phoneArray[i++] = 0;
   ringPhones(phoneArray);
   ...
}
```

因为 ringPhone 仅知道向量元素是 Telephone，它在每个电话上引用 Telephone::ring。一旦企图对 OperatorPhone 调用特殊的 ring 函数，则编译就没有充分的上下文依据来引用。在第 5 章我们将了解到编译是如何使用虚拟函数来自动地提供上下文的依据的。下一节将描述在每个对象中嵌入类型选择域。

4.5 类型选择域

类型选择域（type selector field）是一个基类数据成员，其值表示包括类型选择域的对象类型。在第 5 章所描述的虚拟函数中优先使用类型域（type field）。类型域是高级习惯用法，并在这里概要优先引入使用，又要提供自动的虚拟函数模型的逻辑模式。

例 4.5：带有类型选择域的 Telephone 对象。

```
class Telephone {
public:
```

```
    enum PhoneType (POTS, ISDN, OPERATOR, OTHER} phoneType()
        {return phoneTypeVal;}
    void ring();
    Bool isOnHook();
    Bool isTalking();
    Bool isDialing();
    DigitString collectDigits();
    LineNumber extension();   // 下面不重写
    ~Telephone();
protected:
    LineNumber extensionData;
    PhoneType phoneTypeVal;
    Telephone();
};
class POTSPhone: public Telephone {
public:
    Bool runDiagnostics();
    POTSPhone(): phoneTvpeVal(POTS)...{
        ...
    }
    POTSPhone(POTSPhone &p): phoneTypeVal (POTS)...{
        ...
    }
    ~POTSPhone();
private:
    Frame frameNumberVal;
    Rack rackNumberVal;
    Pair pairVal;
};
class ISDNPhone: public Telephone {
public:
    ISDNPhone(): phoneTypeVal(ISDN)...{...}
    ISDNPhone(ISDNPhone &p): phoneTypeVal(ISDN)...{
        ...
    }
    ~ISDNPhone();
    void sendBPacket();    // 在 B 信道上发送一个包
    void sendDPacket();    // 在 D 信道上发送一个包
private;
    Channel b1, b2, d;
};
```

在例 4.5 中，我们发现了使用类型域的 Telephone 的新的解释，此类型域是对电话对象的不同类型之间的鉴别。这个域存在于所有电话类型的对象之中，所以我们总可以找到一个对象指向 Telephone*并能查明它的类型。注意，Telephone 的每个导出类的构造函数在新的域中存放它的标识。

现在我们可以重新编写 ringPhones。

```
void ringPhones(Telephone *phoneArray[])
{
    for (Telephone *p = phoneArray; p; p++) {
```

```
        switch (p ->phoneType()) {
        case POTS:
            ((POTSPhone*)p) ->ring(); break;
        case ISDN:
            ((ISDNPhone*)p) ->ring(); break;
        case OPERATOR:
            ((OperatorPhone*)p) ->ring(); break;
        case OTHER:
        default:
              error(...);
            }
    }
}
```

固有的 ring 操作现在被独立的电话类所调用，甚至在建立时有一些预见能力。首先已有两种情况恰好已经引用了 p->ring()，因为既不是 POTSPhone 对象有一个惯例的 ring 操作，也不是 ISDNPhone 对象有一个惯例的 ring 操作。但编写此操作确实是项必须的工作。此外，即使 ring 操作被添加到 POTSPhone 或 ISDNPhone 上，这项代码工作照样要进行。

要编写的代码依赖于运行时被操纵的内容的类型域，以决定什么样的函数应当由给定的对象来调用。但是，当对象创建时编译知道一个对象的类，所以编译可以在运行时对每个对象之间相关联的类型信息产生代码。如果在运行时存在这样一个被注释的对象的话，并且成员函数 ring 用于该对象，那么编译可以使用对象类型信息自动地决定调用那一个 ring 操作。这是如何构造 C++虚拟函数的工作。虚拟函数通常优于类型域，并且在第 5 章我们将会进一步加以讨论。

类型域是高级习惯用法，这部分内容在本书后面将加以介绍，诸如模拟多重方法（在运行时选择一个函数则是基于多重参数的类型），这在 9.7 节中将会介绍。另一个例子在 5.6 节中会加以描述。另外，还有可扩充的类型域习惯用法在 Stroustrup[1] 中将会详细加以描述。

下面我们来谈一下类型选择域方法的弱点。当运行时操作符选择在源程序层上由人工管理，在 ringPhone 中用 switch 语句，使程序变得繁琐和麻烦。即使该代码无须重写（如果我们对 POTSPhone 添加 ring 操作），在大多数 C++实现也需要重新编译，同时也需要考虑添加一个新的电话类：ringPhone 也一定要修改，并且与此相关的所有类似函数也要修改。在基类中类型鉴别的枚举需要一个新的元素，并且新类的代码需要在它的所有构造函数中小心地初始化它的类型。所有这些改变必须重新编译和测试。

练　习

4.1 讨论私有继承的使用。

4.2 讨论在 C 中模拟数据抽象和继承的方式。

4.3 编译下面的程序：

```
#include <iostream.h>
class ZooAnimal {
```

1 见 Stroustrup 编写的《C++程序设计语言》一书中第 13 章描述，由 Addison-Wesley，1991 出版。

```
protected:
   int zooLoc;
public:
   int cnt;
};
class Bear: public ZooAnimal {
public:
   int find (ZooAnimal*);
};
int
Bear::find(ZooAnimal *pz) {
   if (cnt) cout << zooLoc;
   if (pz ->cnt) cout << pz ->zooLoc;
}
```

当试图编译此程序时会发生什么？作为与对象和类有关的C++保护模式，这个程序告诉你做了什么？你能防止这个保护模式发生吗？如果能，又该如何防止？

4.4 分析（或编译和运行）下面的程序：

```
class A {
public:
   A (int i = 0) {a = i;}
   A & operator = (A & X) {a = X.a; return *this;}
protected:
   int a;
};
class B:public A {
public:
   B (int i = 0):A(i) {b = i;}
   B & operator = (B & X) {b = X.b; return *this;}
private:
   int b;
};
int main() {
   B b1, b2(2);
   b1 = b2;
}
```

b1.a 的值的结果是什么？如何在使用继承时实现赋值，告诉我们要做些什么？在这个实例中如何修正该问题？

4.5 考虑这样一种软件结构，它为了内存管理和运行时的灵活性，采用了处理/主体（handle/body）习惯用法。即应用类在它自己内部使用辅助类，包含了大部分实现细节。有时对远程应用（distance application）要改变其实现。例如，要改变它的数据结构，该对象的总的函数可以在处理和主体之间加以分割，就好像在继承机制下的基类和导出类之间进行分割一样。

继承可以用两种方式实现：从处理类的外部派生出来，或从内部（“主体”）类中派生出来，对所有的类均要保持有同一信封。研究和讨论这两种选择，并在它们之间权衡。试问还存在其他选择吗？

4.6 试定义一些表示电梯控制系统的类，它是多部电梯在一起运营的系统。对这样系统，应当如何定义这些类的操作？这些类的操作都是用“通用名词”吗？为什么是，又为什么不是？

第5章

面向对象程序设计

面向对象程序设计是一种以封装性和抽象数据类型的抽象性为核心的程序实现风格。程序设计语言和基于类型系统支持的抽象数据类型开发环境，其要素是类型检测并确保双精度添加到生成的机器指令中去，如果是 *a+b* 的话，则其中第一加数要说明为双精度。由类型系统语言初始创建所产生的代码比非类型程序编译后所产生的代码的效率要高，在这中间“类型安全”也是一个需要特别加以考虑的因素。

大多数类型系统在编译及其相关的工具环境中起着一定的作用。上述所提及的因素是传统编译所关注的，并且现在还没有出现这样一种运行程序。但是，编译可以做一些类型转换工作，或者在运行时做一些优化类型系统的可视化方面的工作。考虑下面简单的序列：

```
int i;
double d;
...
d=i;
```

其中编译在运行时要产生整数到双精度的转换代码。整数和双精度之间的相似性使得这样的转换成为合乎逻辑的事，也就是说，这两种类型是耦合兼容的。在 C 语言中，所有类型都是预先定义的，并且编译知道所有类型的操作符的作用，能够解释其上下文，并且能够直接产生代码，执行相关的操作或者在类型之间进行转换。

当用户键入一些新类（其中有程序员定义的新类型，如第 3 章所描述的具体数据类型）和它们之间的关系（如第 4 章使用继承性）时，C++使得采用由人的大脑形成的自然抽象工具和以层次结构组织为原则，来构造一个系统成为可能。这些层次结构和类型编译系统不可能知道，所以需要帮助用户在相似类之间提供一定程度的耦合兼容。

在大型系统中许多不同类的层次结构在认知上是可管理的，不可管理的是极少数的个别类。为了利用层次结构这种特性，程序员需要沿着层次结构的顶层做有效的工作，而无须深入研究导出类。这个原则同样也适用于设计和编码。根据高层抽象 SingnedQuantity 编写代码，实际上在运行时可以处理 Complex 类和 InfinitePrecision 类的值，其中后面两个类是在 SingnedQuantity 继承树控制之下的。如果程序员写

```
void afunction(SignedQuantity &c, SingnedQuantity &d) {
    ... c+d ...
}
```

那么，给相关的类中添加操作必须在运行时按照上下文来选择，这就大大超越了编译时的类型系统。

基于性能和类型安全方面的考虑，运行时支持的正是编译时未能支持的那些部分，这就是面向对象程序设计的本质。新的类型系统构成了运行时类型支持的抽象数据类型的封装和继承的层次组织结构。

C++必须在编译时在类型系统部件之间取得平衡，以便维护代码效率、提高运行时部件的灵活性以及提高效率方面的耦合兼容性。一方面，C++语言提供了面向对象程序设计所需要的基础；另一方面，也要维护与C语言兼容的文化内含结构。

面向对象程序设计在某种意义上是使用间接手段的一种程序设计技巧。有两点值得注意，一是语言对这些技术支持的重要性；另一个是设计原则的重要性。这些注意事项告诫我们要增加灵活性和更高级的抽象需要添加一些习惯用法，但程序员和设计者必须记住设计技术是带有习惯用法最具效率的杠杆。

下面我们将讨论这些习惯用法。

5.1 C++运行时类型支持：虚拟函数

类继承赋予程序员组织代码一种支持软件重用新方式的能力。基类将它的导出类中共性的代码纳入其中并加以托管，导出类则可扩充并改进基类的行为，或具体化基类的抽象行为。习惯上把基类指针的变量作为它的导出类对象的地址，并且引用该对象的任意一个公共（基类）成员函数。这就提供了多态性，即相同的指针可用于几个不同形式的对象操作。程序员可以说明一个在继承树中有效地指向类集合中任一对象的指针。使用这样的指针，则所有这样的对象就能够平等地被操作，无须了解具体的类，每个类都可根据它的通用类型、层次结构的基类进行处理（如图5.1所示）。

```
class Employee{
public:
  double salary() {return sal;}
  double computeRaise() {return 25;}
  Employee(double salary){sal = salary;}
private:
  double sal;
};
```

```
class Manager: public Employee{
public:
  double computeRaise(){return 100;}
  Manager(double salary):
    Employee(salary){ }
};
```

```
class SecurityGuard: public Employee{
public:
  double computeRaise(){return 50;}
  SecurityGuard(double salary):
    Employee(salary){ }
};
```

```
Manager *boss1 = new Manager(2000);      // 有效
double boss1Salary = boss1 ->salary();  // 产生 2000
Employee *boss2 = new Manager(2300);     // 有效
double boss2Salary = boss2 ->salary();  // 返回 2300
double boss1Raise = boss1 ->computeRaise();    // 产生 100
double boss2Raise = boss2 ->computeRaise();  // 产生 25
```

图5.1 使用继承的多态性的一种弱形式

当指针的类型精确地与对象的类型相匹配时，返回正确的结果。当通用基类的类型指针用于一个导出类地址时，则仅引用基类函数。驻留在对象类的其他同名的函数的指针是不可见的。

继承允许应用于对象公共的成员函数集合，而这些对象类在继承层次结构中作为同类彼此之间

是相关的。但是，如果继承仅用于支持多态性，则仅扩充了以类共享实现的那些函数。如果在基类成员函数实现上导出类有它自己的变化，当通过类变量说明的对象来引用函数时，则继承就不产生已挑选过的导出类版本。指针知道基类的类型，但不知道其结构，或者甚至不知道导出类的存在。在图 5.1 的上下文中，考虑用 Employee 类的薪水管理程序。如果程序一般地理解 Employee，而 SecurityGuard 和 Manager 不知道 Employee，仅知道是在 Employee 层次结构中，就对任一对象进行缺省处理。

我们要求基类有健壮性接口，该接口塑造了抽象行为的表示、名字表示、返回类型和它的成员函数的参数类型。基类的公共成员函数要么是导出类接口的一部分，要么用比具有相同逻辑行为更合适的函数的导出类来替代。所有导出类行为用于所有导出类的对象，所以该基类表示了所有导出类的行为。

在第 4 章中，不同类型的电话定义了不同的 ring 成员函数。导出类可以重写它的基类成员函数的实现，即可以提供它自己函数的实现，出现在其基类公共接口上。重写的基类成员函数的名字与导出类成员函数的名字相同，这是改善其实现效果的较好的方式，因为它们用名字维护了该函数的基本含义。ring 操作符给不同类的电话应用赋予了相同的含义，虽然根据不同的电话类型其实现细节不尽相同。

我们要求编译对这些对象的可交换性尽可能地透明。编译知道命名合适的函数出现在基类接口中，但我们要从合适的导出类中自动地调用这个函数，当然，此导出类的对象是在初始时创建的。我们要用对象来确定此函数的选择，而不是习惯地用指针的地址来说明。其结果程序员可以真正地、平等地处理这些类的所有对象，而且可以产生正确的结果。如 Manager 若在管理上获得加薪，即使加薪的管理程序 computeRaise 仅知道在编译时一般的 Employee 的情况，也可以正确地加以处理。

这种能力称为运行时操作符标识(identification)，并与继承性一起提供给设计者和程序员创建高层软件部件灵活的工具箱的多态性。这种在具体数据类型所使用的一致性能力被称为面向对象程序设计，它是在 C++中最直接支持的面向对象设计。

运行时操作符标识仅用于引用或者指向类对象的指针。为什么呢？因为只有对说明为指针或引用的变量才可能涉及到一些后继的导出类对象，或者涉及到它们的类对象。在这种情况下，一个合理的建议是仅使用指针的，抛弃其类型信息：如果要在实例上直接引用一个成员函数，那么必须说明这个对象类，并且这样的引用是在编译时进行的。但如果指向对象的指针的存在距离初始创建点比较远，那就不必要用户这么做了。

让我们回顾一下在第 4 章中的 Telephone 的例子。实际上，只对少数地方的修改是必须的。在例 5.1 中我们再回忆一下。

例 5.1：Telephone 例子。

```
class Telephone {
public:
    void ring ();
    Bool isOnHook ()  isTalking (), isDialing ();
    DigitString collectDigits ();
    LineNumber extension ();
    ~Telephone ();
protected:
    Telephone ();
```

```
    LineNumber extensionData;
};
class POTSPhone : Public  Telephone {
public:
    Bool runDiagnostics ();
    POISPhone ();
    POISPhone (POTSPhone&);
    ~POTSPhone ();
Private:
    Frame frameNumberVal;
    Rack rackNumberVal;
    Pair PairVal;
};
class ISDNPhone : public  Telephone {
public:
    ISDNPhone ();
    ISDNPhone (ISDNPhone&);
    ~ISDNPhone ();
    void sendBPacket (), sendDPacket ();
private:
    Channel b1, b2, d;
};
class Operatorphone : public  ISDNPhone {
public:
    Operatorphone ();
    Operatorphone (Operatorphone&);
    ~Operatorphone ();
    void ring ();    // 专门的响铃所需要的操作符
};
class PrincessPhone : public  POTSPhone {
  ...
};
```

注意，我们使用的是类版本，不需要嵌入在第 4 章中介绍过的“类型标识”信息。在这里，我们不需要这么做，编译可自动地加以处理。我们用 virtual 说明一些基类成员函数，并能在导出类中提供定制的相同的函数版本。编译引用虚拟成员函数所调用的代码，根据对象的类型在运行时选择正确的属于该函数的导出类版本。每当从虚拟函数的类中构造一个对象时，编译就把它存放在该类的“类型域”中（详细实现需要交叉编译）。而且，编译这么做比手工做有着更高的效率。例如，当需要添加一个新的导出类时，手工管理类型域的程序需要分布地编辑并进行重新编译；当使用虚拟函数时，所有在大多数 C++环境中所涉及到新的导出类的工作均自动地重新编辑和编译。手工管理还可能导致程序产生不同的解。

使 Telephone 类面向对象的第一步是将它的成员函数说明为虚拟的。使基类函数虚拟就是告诉编译器，这些函数的行为和实现与正在操作的类有关，而且这个基类不能预见所有导出类的需求，这就给我们提供了导出类中这些函数的新的实现。也就是说，可以存在大量的给定的虚拟函数的不同版本，它在基类的层次结构中与每个类一样多。

通常情况下，基类定义了虚拟函数的主体(body)。如果未选择重写其实现的话，则可以作为缺省由导出类继承；如果该导出类不重写这个函数，则这个导出类的行为就好像基类函数一样，是基

类接口的一部分；如果在一个或多个导出类中重写了该函数，那么导出类版本就重写了基类的版本。

虚拟函数使编译在运行时支持导出类对象的操作。当在基类中引用一个说明为 virtual 的函数时，该函数就调用导出类中适合的函数。我们要求这种操作用在合适的对象类型中，不涉及到习惯用指针引用的那种对象类型（如图 5.2 所示）。

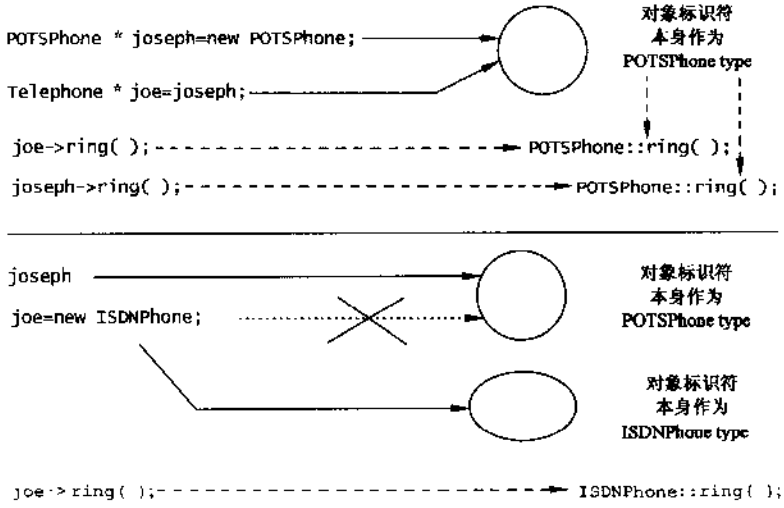


图 5.2 虚拟函数运行时标识

当函数是虚拟时，操作符的标识引用的是对象正在指向的类型，而不是指针的类型。

下面是基类如何用 virtual 说明的表示：

```
class Telephone {
public:
   virtual void ring ();
   virtual Bool isOnHook ();
   virtual Bool isTalking ();
   virtual Bool isDialing ();
   virtual DigitString  collectDigits ();
   virtual ~Telephone ();
   LineNumber  extension ():
protected:
   LineNumber  extensionData;
   Telephone ();
};
```

virtual 关键字不是随便添加的，而是添加在基类需要的地方。virtual 关键字必须添加在类说明中说明函数的位置上，而不是函数定义的那个位置上。

在基类设计中，我们可以创建一些在导出类中永远不被重写的函数。例如，成员函数 isOnHook，isTalking 和 isDialing 是原始成员函数，它们可直接作为基类成员函数来实现，无须导出类协助。即这些成员函数不仅是位置占位符，而且对所有导出类来说，它们也提供了合适的缺省行为。经验告诉我们，当系统在演变之时，基类成员函数经常需要在导出类中重写，以便在初始说明为虚拟的地方保持一致。通常在说明了函数虚拟以后，函数调用的花费大约增加了 15%。如果函数调用可以获

得程序执行 10%的好处的话，则使用虚拟函数将下降大约 2%的性能。但是，虚拟函数需要与不采用虚拟函数而需要更改的实现作比较。如在 C 中若喜欢用 switch 语句或 if 语句编写程序的话，而 C++程序在运行时用虚拟函数在 switch 语句或 if 语句中间作替换，我们将会发现，某些应用程序（例如，用 C++编写的电路设计解释器[1]）用虚拟函数比用 switch/case 方法要快得多。

现在，我们可以使用新的行为来重写应用程序，这里是一个老的 ringPhones 函数：

```
void ringPhones (Telephone *phoneArray [])
{
   for (Telephone *p = phoneArray; p; p ++ ) {
      switch (p ->phoneType () ) {
      case POTS:
         ((POTSPhone *) p) ->ring (); break;
      case ISDN:
         ((ISDNPhone *) p) ->ring (); break;
      case OPERATOR:
         ((OperatorPhone *) p) ->ring (); break;
      case OTHER:
      default:
            error (...);
      }
   }
}
```

现在更合适的简单版本是：

```
void ringPhones (Telephone *phoneArray [])
{
   for (Telephone *p = phoneArray; p; p++) p ->ring();
}
```

如果 p 指向 OperatorPhone 对象，则 p->ring()将引用 OperatorPhone::ring()。如果在这个 Telephone 层次结构中 p 指向任一其他类型的对象，那么 Telephone::ring()将替代此引用，因为没有一个其他的类重新定义过电话呼叫的缺省行为。

注意，这项工作仅是在基类中函数被说明为虚拟时才执行。下面代码就发生了这类问题：

```
Telephone *digitalPhone = new ISDNPhone;
...
digitalPhone ->sendDPacket ();    // 错误，不能找到此函数
```

这个例子是错误的，是不能执行的，因为 sendDPacket 没有在基类 Telephone 中加以说明。即使 sendDPacket 在它自己的类中已经说明为虚拟，它仍不能执行 ISDNPhone 的导出类中的虚拟行为。若在运行时产生操作符标识的话，则成员函数必须被虚拟化。一个成员函数可以用两种方式之一而成为虚拟的：一种方式是，如果在类中用指向引用操作的指针来说明虚拟，那该成员函数就是虚拟的。如果同名、同类型、同参数的成员函数出现在顶层的继承层次结构中（如图 5.3 所示），那么这个成员函数也是虚拟的。此虚拟的函数是沿着继承层次结构导出类一层一层地继承下来的。

1 Russo, V. F. et al. "A C ++ Interpreter for Scheme". Proceeding of the C++ workshop. USENIX Asscociation publishers (Oct. 1988).

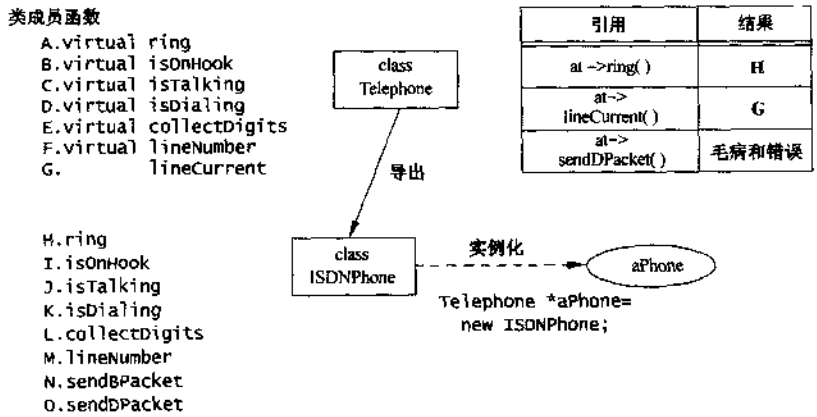


图 5.3 函数虚拟的继承

另一种方式是采用初始创建一个说明用 virtual 关键字，并且是与成员函数不同的函数。虚拟成员函数可以认为不是一个函数而是作为函数名字的说明。初始（非虚拟的）成员函数说明是函数，在编译时或链接编辑时，它们直接捆绑在函数主体上。虚拟函数不这么做，当虚拟函数名字在运行中被引用时，它的名字习惯捆绑在被调用的函数上。就这种含义而言，虚拟函数在 Smalltalk 中类似于方法选择器，而成员函数主体类似于 Smalltalk 的方法。

5.2 虚拟析构函数

注意，析构函数 Telephone::~Telephone 也是虚拟的，考虑下面的代码：

```
class Telephone {
    ...
    ~Telephone ();
    ...
};
Telephone *digitalphone = new ISDNPhone;
...
delete digitalPhone;
```

引用 delete 是删除 ISDNPhone 对象，这就意味着如果 ISDNPhone 的构造函数分配的资源被 ISDNPhone 析构函数释放的话，则它们就变成了“垃圾”。

如果基类 Telephone 的析构函数被说明为虚拟的话，那么在运行时此代码对操作在该对象上的类型寻找合适的析构函数：

```
class Telephone {
    ...
    virtual ~Telephone ();
};

Telephone *digitalPhone = new ISDNPhone;
```

```
...
delete digitalPhone;    // 正常工作，调用 ISDNPhone :: ~ISDNPhone ()
```

然后，再引用对象自己的析构函数，产生合适的清除动作。当然，基类析构函数 Telephone::~Telephone() 将仍旧立即调用后面紧跟着的 C++析构函数行为。

5.3 虚拟函数和作用域

函数虚拟可以改变它的作用域，这可能在程序语义上会产生微妙的变化，这些语义在构建继承层次结构中是相当重要的。

首先，其重要性在于记下任一函数接收运行时操作符标识处理，必须达到精确地与在基类中虚拟函数有相同的特性。一个函数的特性包括构造它的名字、参数类型规格说明次序和函数的返回类型（在 C++中，返回类型不是分辨调用重载函数的特性的一部分，但仍是函数特性的一部分）。改变虚拟函数的返回类型是非法的，例如下面的编码：

```
class Number {
Public:
    virtual float add (const Number&);
    Number(double);
    ...
};
class BigNumber:public Number {
public:
    virtual double add(const Number&);  // 非法
    BigNumber (double);
    ...
};
```

这种改变是非法的，因为在一些对象上 add 操作的引用会产生在编译时不知道的表达式类型上，所以它不可能产生处理返回值的代码：

```
void someFunction (Number &num) {
    Number a = 10, *num;
    ... num ->add(a) ...          // 是 double 还是 float?不知道!
};
```

另外一种“破坏”虚拟函数的行为是改变导出类中函数的参数列表。这其实是合法的操作，但产生了两个性质截然不同的函数，相互之间没有联系：在顶层的继承层次结构上，一个函数隐藏了另一个同名函数。例如，考虑例 5.2 中的程序，其中 BigNumber::add(double)完全隐藏了在基类中同名的一个函数，所以它难以获得用户的导出类。如果 BigNumber::add 说明为 virtual，则可以得到正确的结果。

例 5.2：标识改变破坏了虚拟继承。

```
class Number {                    // 基类
public:
    virtual void  add(int);
    ...
};
```

```
class BigNumber :public Number {  // 导出类
public:
    void  add (double);
    ...
};
int main() {
    Number *a;
    BigNumber *b, bo;
    ...
    a ->add(1);                 //Number::add(int)
    a ->add(3.0);               //带强制 int(3.0)的 Number::add(int)
    b ->add(2);                 //带提示 double(2)的 BigNumber::add (double)
    b -> add(2.0);              //BigNumber::add(double)
    b ->Number::add(7.0);       //带强制 int(7.0)的 Number::add(int)
    bo. add(8);                 //带提示 double(8)的 BigNumber::add(double)
    bo. add(9.0);               //BigNumber::add(double)
    bo. Number::add(9);         //Number::add(int)
    bo. Number::add(10.0);      //带强制 int(10.0)的 Number::add(int)
    return 0;
}
```

作用域和虚拟函数可在其他微妙的方式中相互影响。考虑下面的代码：

```
#include<iostream.h>
class A {public: virtual void f (int); };
void A::f(int) {cout << "A::f \n"; }
class B: public A {public: void f (double); };
void B::f(double) {cout <<"B::f \n"; }
class C: public B {public: virtual void f(int); };
void C::f(int) {cout <<"C::f \n"; }
int main ()
{
    A *a;
    B *b;
    C *c = new C;
    c ->f(2);          //C::f
    c ->f(2.0);        //带强制的 C::f
    b = c;
    b ->f(2);          //带提示的 B::f
    b ->f(2.0);        //B::f
    a = c;
    a ->f(2);          //C::f
    a ->f(2.0);        //带强制的 C::f
    return  0;
}
```

这里，虚拟函数定义在基类 A 的根部，有不同特性的非虚拟函数定义在中间类 B 中，在叶子上类 C 重新定义了虚拟函数。值得注意的是，在类层次结构中高层的虚拟函数可被层次结构的中间成员函数隐藏。一种方式认为这是类 C 的一个对象带有 C::f(int)，但 C::f(int)的可见性改变了指针地址的类型。这与我们的直觉相反，希望虚拟函数无论如何在引用时要确保被调用的对象有相同的函数。现

在，大多数 C++系统警告出现在上述继承结构中具有同名的虚拟函数这样的重载函数隐藏。

5.4 纯虚拟函数和抽象基类

我们再来看一下第 4 章的 Telephone 例子，其中 protected 节的构造函数不可能创建 Telephone 实例（见第 4 章 4.2 节的例子）。实例只能通过导出类来创建：就像 POTSPhone（普通电话服务），VideoPhone（视频电话），ISDNPhone（集成服务数字网络电话）那样。我们不能构建实际上的抽象 Telephone！为什么呢？因为我们不能编写通用的打电话的 ring 成员函数代码，而只能编写所知道的具体电话类型的打电话代码。POTS 电话获取呼叫铃响电压；数字 ISDN 电话获得产生呼叫铃响和亮灯的信息。

C++有简明的获取成员函数语义的语法，诸如在函数说明本身的 ring。首先，我们识别出这样的函数应该是虚拟的：我们要求系统软件使用所有不同类型的 Telephone 对象，使之能够引用它们的 ring 成员函数，而不需要知道其实现细节和精确的类型。第二，我们知道不能编写一个通用的、对这样一种函数的缺省主体的代码，所以我们不得不把函数的主体约束在空（null）指针上。

```
class Telephone {
public:
    virtual void  ring () = 0;          // 纯虚拟函数
    Bool isOnHook ();
    Bool isTalking ();
    Bool isDialing ();
    virtual DigitString collectDigits ();
    LineNumber  extension ();
    virtual ~Telephone ();
    Telephone ();                       // 又一个公有函数
    ...
};
```

我们把这样的函数称为**纯虚拟函数**。通常，纯虚拟函数只表明了电话的位置标志，一般情况下可以呼叫，虽然我们不能表示如何执行的状态。“纯”的含义表示的是没有函数主体，而仅有接口和纯的特性。

现在要创建 Telephone 类的对象是不可能了，因为编译不知道做什么，甚至是否 ring 操作在类上执行与否编译也不知道。这样的类称为抽象基类。在定义某些行为并且无须定义这些行为实现的时候，就这种意义上来说，它是抽象的。把定义成员函数集合的责任强加在它的导出类上。就此而言，类定义了一种抽象数据类型，无须定义其细节。如果所有它的成员函数是纯虚拟的，那么抽象基类就是抽象数据类型（ADT）的规格说明。

纯虚拟函数的说明比第 4 章 4.2 节 Telephone 例子隐含的构造函数的说明要更多一些。首先，说明了不能被实例化的类、形式参数类型和函数返回类型。隐含的构造函数也有这些特性，从导出类成员函数内可以创建具有窥视孔的基类对象。第二，如果导出类有任一实例的话，则纯虚拟函数说明必须在导出类中重写。

一个纯虚拟函数可以提供一个主体，使用作用域分解的操作符可以调用带有主体的纯虚拟函数：

```
#include <iostream.h>
class Base {
public:
   virtual void  pure () = 0;
};
class Derived: public Base {
public:
   void  pure () {cout <<"Derived::pure" << endl; }
   void  foo () {Base::pure (); }
};
void Base::pure () {cout <<"Base::pure"<<endl; }
int main () {
   Derived d;
   d.Base::pure ();
   d.foo();
   d.pure();
   Base *b = &d;
   b ->pure ();
   return 0;
}
```

当运行时，该程序产生以下输出：

```
Base::pure
Base::pure
Derived::pure
Derived::pure
```

如果删去 Base::pure 的主体，则产生链接-编辑-时间的错误结果。

这可以应用于 Window 的例子。Window 可以坚持主张它的导出类提供自己的 clear 版本，而且仍能对所有窗口类型提供公共的基本 clear 功能。例如，当导出类 clear 成员函数作为最终动作引用时，Window::clear 可以对屏幕中心的游标进行定位。

5.5 信封和信件类

处理/主体类的习惯用法支持多种设计技术，可提供较好的灵活性、较高的性能，并能减少传统规范格式变更的影响。信封/信件是使用一对类的技术，一个动作在外部（“处理”类），它是可见部分，而另一个动作是在内部（“主体”）类，其中包含了实现细节。处理是对主体所要求的内部事务的转发。这两种形式一起构成了复合对象。这里，我们将介绍处理/主体习惯用法的扩充，称之为信封/信件习惯用法，这种习惯用法增强了上述提供的处理和主体的灵活性以及封装性。

信封/信件类习惯用法是处理类习惯用法的一种特殊情况，当处理类和主体类共享同一行为，而不是当主体类行为特化或增大处理类行为时才使用。当且仅当使用此种方式时，信封称为处理类，而信件称为主体类。这两种类的关系和基类（信封类）与导出类（信件类）的关系如出一辙，但比继承更具运行时的灵活性。信件类由它的信封类导出，而它的实例也包含在信封类的实例中。

信封/信件类通信通常比层次结构中的类更要自主。大多数信件类只通过它们的信封类存取，这与在继承层次结构中的类完全相同，但在程序代码的导出类构造函数引用中，普遍存在着例外处理

（exception treatment）。信件类构造函数引用必然限定在信封类代码区域内。

当你使用这种习惯用法时:有时，我们要求比C++编译时联编更好的灵活性。这样的灵活性可以通过取代一些传统编译时函数的做法，诸如用对象/类关联来替代。信封/信件类能够提供比带有虚拟函数的继承更优的多态性和运行时的类型支持。这就能简单明了地概括相关类包的用户相互作用，它也能减少代码变化和重新编译的影响。

信封和它们的信件之间的关系可以认为存在两种方式。第一种方式是，信封可以预见为一些功能上协作的信件对象的委托。第二种方式是，信件可以作为信封的雇用（使用），逻辑上封装在信封内。第二种模式特别适合强制的封装性。

下一节将看到信封/信件类提供了比单一的、编译时类的“本地”C++更强大的面向对象程序设计习惯用法。

信封类和委托多态性

在纯面向对象程序语言(如 Smalltalk)中，运行时变量被捆绑在对象上，其作用就像标号一样：将变量捆绑在对象上就像把标号粘贴在对象上一样。在这些语言中，赋值相当于标号从对象剥离下来并粘贴上另一个标号。在这些语言中变量几乎没有或者无类型信息与之相联系，并且在编译时几乎未做强制类型耦合工作。当然，对象本身具有操作、成员函数以及由类型信息所组成的属性，所以类型检测仍能在运行时执行。具有这种特性的语言有一些缺点，因为类型可能在运行时突然出现，但这样的语言提供了原型和程序演变的高层灵活性以及高级内存管理技术的基础。C++类型模型不具有这种运行时的灵活性，这有两个原因：一是早期就进行了符号联编的访问；另一个是早期就进行了强制类型的兼容工作。

在C和C++中，变量是用对象标号替代的地址和位移的同义词。赋值并不意味着重新标号，而是用新的内容重写旧的内容。我们用人工添加间接层的方法来补偿，通过指针来访问对象。虽然变量名仍与一大片存储区有关，但存储区“仅仅”是一个指针，所以，名字和“真实”对象之间的关联很容易用重写该指针的方法来加以改变。但是，有时带有指针的程序设计难以处理。内存管理会变得更加困难：悬挂指针可能发生。通过指针访问所有对象，会造成在使用一些程序设计语言特性时不太方便，特别是在使用操作符重载的时候：

```
class Number {
public:
    Number();
    virtual Number *operator *(const Number&);
    ...
};
class Complex: public Number {
public:
    Complex (double, double);
    Number *operator*(const Number&);
    ...
};
class Integer:public Number {
public:
    Integer(int);
    Number *operator*(const Number&);
```

```
    ...
};
Number *numberVector[5], *numberVector2[5], *productVector[5];
    ...
numberVector[0] = new Complex(0, 10);
numberVector[1] = new Integer (0);
numberVector[2] = 0 ;
numberVector2[0] = new Integer(10);
numberVector2[1] = new Complex(10, 10);
for (int i=0; numberVector[i] && numberVector2[i]; i ++) {
    // 糟糕(但必须的)的语法:
    productVector [i] = *numberVector[i] **numberVector2[i];
}
```

第二个复杂性是C++编译要确保对象能够满足成员函数变量的访问，这需要通过编译的类型系统来强制执行。就这种意义而言，“类型系统”在应用中的概念抽象只有一些细微的关系，可以作为正规类型的程序反映在它的类结构之中（参见6.1节）。类型系统已经证明对防止错误是有用的。C++类型系统在编译时能够捕获到类型的错误，C++类型系统保存了类结构的解释，并用推理方式看待设计抽象之间的关系：它捕获所有应用领域类型的真实错误，但也可能屏蔽了一些有效应用程序的使用。

松散的捆绑变量-对象和运行时符号语言的强类型系统减轻了程序员在类型之间说明兼容性的繁重工作。我们经常要处理类的互换性聚集，把这些类的所有对象看做是属于同一类的。虚拟函数提供了这种功能性指针和引用变量，但是指针是脆弱的——现在已经把它称之为“数据的GOTO”，并且指针对重载操作符综合得不太好。在上述Number的例子中，指针允许我们处理像Number那样的大量的数，但在不灵活的语法处理上花费是巨大的。我们要做的是处理像Number这样的事情，并且让每个对象本身执行有关如何处理这些事情的专门的技能：

```
int main () {
    Number c(1.0, -2.1);
    Number r(5.0);
    Number product = c * r;
    r = 0;
    return 0;
}
```

这里出现的一个问题是，该对象在运行时必须判定是属于哪一个类：c变成了Complex，r变成了DoublePrecisionFloat，而被除数变成了假定要变成的一个数（在理论上，前两个可以在预先编译时执行，第三个通常必须在执行期间才操作）。不同类的对象采用不同的存储区，所以一个对象可以变更它的类，但必须改变它的大小，编译要知道对象的大小数量。对象捆绑在虚拟函数上，是在对象初始化期间执行，并且不能改变，即按对象的方式改变它自己的类型。另一个问题是一旦对象类被设定，这个对象可能需要“变更”它的类到底应该会是什么样的内含。例如，product的初始定义推荐它应该是一个复数，当设定为零后，可能就不需要保持它的Complex标识了。

我们的确需要了解运行时对象和它的类之间的关联。此外，我们还要了解其具有松散关联的对象的类的变迁。即，允许执行中“采用不同类型”这样一种可变性。这样的灵活的数包（number package）实现的类聚集如图5.4所示，并由例5.3和例5.4列出了部分实现的细节程序。

为了维护语义的合理性，作为数的伪装实例的所有类必须有一个 Number 的特性。在 C++中，我们表达这样的类必须采用抽象类 Number。数字类型的层次结构可以构造在 Number 之下，这些类的对象一定包装在“信封”类中，以保护其值的标识（或地址）——以及潜在的类型——用操作来改变它的应用。这个“信封”有对象的相同的特性，并且同一类的 Number 既能作为“信件”类的抽象类，又能作为维护用户表示细节的“信封”类的双重责任。该模型如图 5.4 所描述，其代码由例 5.3 表示。

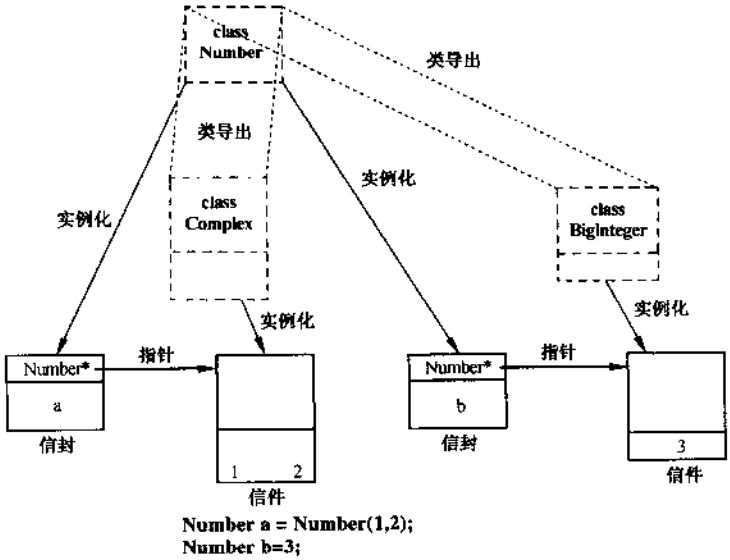


图 5.4 多态的 Number

例 5.3：Number 类部分实现代码。

```
struct BaseConstructor {BaseConstructor (int = 0) { } };
class Number {
public:
   Number () { rep = new RealNumber (0.0);}
   Number (double d) {rep = new RealNumber (d);}
   Number (double rpart, double ipart) {
      rep = new  Complex (rpart, ipart);
}
Number operator = (const Number &n) {
   n.rep ->referenceCount ++;
   if (-- rep ->referenceCount == 0) delete  rep;
   rep = n.rep;
   return *this;
}
Number (const Number &n) {
   n.rep ->referenceCount ++;
   rep = n.rep;
}
virtual Number operator + (const Number &n) {
   return rep ->operator + (n);
```

```
}
virtual Number complexAdd(const Number &n) {
    return rep ->complexAdd(n);
}
virtual Number realAdd(const Number &n) {
    return rep ->realAdd(n);
}
void redefine (Number *n){
    if ( -- rep ->referenceCount == 0 ) delete rep;
    rep = n;
}
...
protected:
    Number (BaseConstructor) {referenceCount = 1;}
private:
    union {
        Number *rep;
        short referenceCount;
    };
};
```

例 5.4：Complex 类的部分实现代码。

```
class Complex:public Number {
public:
    Complex (double d, double e):Number (BaseConstructor ()) {
        rpart = d; ipart = e;
        referenceCount = 1;
    }
    Number operator + (Number &n) {return n.complexAdd(*this);}
    ...
    Number realAdd(Number &n) {
        Number retval;
        Complex *c1 = new Complex (*this);
        RealNumber *c2 = (RealNumber*) &n;
        c1 ->rpart += c2 ->r;
        retval.redefine (c1);
        return retval;
    }
    Number complexAdd (Number &n) {
        ...
    }
private:
    double rpart, ipart;
};
```

下面是 3.5 节讨论过的引用计数对象的习惯用法，Number 对象的大多数内存智能管理是在信封类中，而应用语义是在信件类中（见例 5.3 和例 5.4），构造函数建立在带有参数的信件对象的适合的种类上。赋值和拷贝构造函数使用 3.5 节介绍过的引用计数习惯用法，它们是交织在一起的引用计数是在联合中，在信件类中联合的空间作为计数域来使用，而在信封类中又作为信件指针来使用。从 Number 到信件内部维护都由操作来实施，operator+和其他操作符的实现见例 5.3。

例 5.3 的 Number 类既可用于用户所看见的“通用数”的对象类，又可用于信件类的基类；它的一些成员函数给导出类提供了专门的服务。专用的构造函数 Number::Number(Base Constructor)初始化了 Number 的实例，是信件类的基类格式的一部分。它取代了缺省构造函数 Number::Number()，在 Number::Number 内创建 Number 导出类的新实例中避免了无限递归的发生。

成员函数 complexAdd，realAdd 和 redefine 是特殊应用的成员函数，给导出类提供了服务；但是因为它们是交叉引用的实例，所以不在保护节（protected）中。任何 Number 对象必须准备接受任何其他数（Number）对象的“种类”的实例，作为操作的一种操作数。考虑下面的代码：

```
Number aComplex(1.0, 2.0);
Number aReal(3.0);
Number result = aComplex + aReal;
```

当 aComplex 的 operator+操作用 aReal 作为参数引用时，就转交了对它的信件对象的要求。这是用 aComplex 作为操作数的 Complex 的 operator+引用，并用 aReal 的值作为参数。Complex::operator+依次引用(在它的参数上)complexAdd 操作，传递*this 参数（aComplex 值）。现在的控制是在 Number 的 complexAdd 内部，刚好转交了对 RealNumber 的 complexAdd 的调用。因为对两个对象进行了“友好”的访问，所以 RealNumber 的 complexAdd 成员函数生成了合适的结果类型并正确地返回之。

模拟“虚拟构造函数”的概念

面向对象设计一个重要的设计原则是每个类应该知道如何维护它自己的事务。面向对象程序设计的重要基石是多态性，它源自于继承和虚拟函数的组合。将继承和虚拟函数组合在一起的用户可以通过基类定义的接口只与对象进行通信。在运行时，编译提供成员函数调用合适的导出类成员函数的分配技巧。C++构造隔离了用户想要知道的数（Number）的细节和导出类的一些特性，并屏蔽了这些类的实现细节。

在 C++中，有一个例外是它并未达到纯面向对象语言的基本要求。也就是说，C++不是纯面向对象语言，而是混合型面向对象语言。考虑一个从 BigNumber 和 Complex 导出的 Number 类，其中大多数导出类操作（成员函数）在基类中也作为虚拟操作出现。带有对 Number 指针或引用的用户可以对 Number 类、BigNumber 类和 Complex 类的对象进行捆绑，并能用 Number 的虚拟函数与这些类很好地进行交互。用户不需要了解 Number 类层次结构的细节，平等地处理所有实例，根据 Number 继承特性进行访问。

如果用户不能生成 new BigNumber 和 new Complex 的话，那么这就违背了用户不应知道有关导出结构或低层类的结构细节的原则。所以指针必须来自于 Number/BigNumber/Complex 组合本身的内部。我们推断 Number 构造函数创建了 BigNumber，因为它的参数比某些界限值要大，或者也许 sqrt 操作当引用负数值时在 Number 内创建了 Complex。

当用户假定了 new Number 时，那么我们可以断定 Number 的构造函数编造了一个 BigNumber 和一个 Complex，并安排了 operator new 返回值指向它们之中的一个。但这是不可能的，我们考虑下面的代码：

```
Number :: Number () {
        if (some condition)
            this = new BigNumber:        // 非法
}
```

在C++中，对this赋值是非法的。赋值的典型使用是以operator new更多的限制能力来满足的。

解法：我们可以用第三方抽象作为代理，去执行代表客户创建对象的Number/BigNumber/Complex事务。用做信封类的Number类对象可以完全扮演这种代理。每个Number实例指向在一个导出类中所创建的信件对象——例如，是BigNumber或者是Complex。这个指针说明为Number*：

```
class Number {
private:
    Number *rep;
    ...
}
```

Number的成员函数使该对象指向私有rep域：

```
public:
    Number operator *(const Number &n) {
        return rep ->operator *(n);
}
...
```

信件的成员函数说明与Number的成员函数以及它们的基类完全一样。私有rep域用信封构造函数来初始化，其构造既可能是BigNumber，又可能是Complex，这依赖于某些条件，并且把指向结果对象的指针存放到私有rep域中。

以Number类为例，我们把Number类存储在输出流上（C++通用的iostreams库的ostream类型）。但我们也希望从istream（输入流）中初始化Number，即，赋予Number一种能力去读它本身，在这个过程中创建一个新的对象。所以存在下面这种形式的构造函数：

```
class Number {
public:
    ...
    Number (istream& s) {
        char buf [10];
        S >> buf;
        switch (numParse (buf)) {
        case COMPLEX:
            rep = new  Complex(buf);
            break;
        case FLOAT:
            rep = new DoublePrecisionFloat (buf);
            break;
        ...
        }
    }
 Private:
    enum NumType {COMPLEX, FLOAT, BIGINT};
    NumType numParse (char *);        // 确定了数的类型
    Number *rep;
};
```

当然，实际上rep是在numParse的主体内指定的，以避免在所有数字类中rep和Number::Number (istream&)之间重复构建。

现在类出现了构建对象的不同种类的情况，其特性由构造函数所提供的上下文来决定的（这里，是由一个文件或其他源头的字节流的内容决定）。

这个给定的类在运行时的灵活性是通过虚拟函数赋予对象的，能力来自于所熟知的虚拟构造函数。虚拟构造函数进而模拟 C++中整个基类的类，使用 C++就好像是使用一个单一层次的结构系统一样。关于这种灵活性的深层次的讨论见第 8 章。

当使用这种习惯用法时：当对象的类型需要从对象构造的上下文来决定时，就要考虑到使用虚拟构造函数的习惯用法。使用虚拟构造函数的例子有构建含有类型和尺寸的窗口对象，作为屏幕函数使用；或者创建交互式对象和无格式文件输出使用。

下面的例子是在文本语法树中使用了表示原子（不可分的词法记号）的 Atom 类的体系结构。Atom 担负了简单语法系统的许多词法工作。这里是基类的定义，其中所有的成员函数都说明是虚拟的：

```
class Atom {
public:
   Atom () { }
public:      //所有 Atom 特性的联合
   virtual ~Atom() { }
   virtual long value()       {return 0; }
   virtual string name()      {return String ("error"); }
   virtual operator char ()   {return 0; }
   virtual Atom *copy () = 0;
};
```

注意，Atom::copy 是纯虚拟的：有一个紧跟着的“=0”的规格说明。这些类是直接从 Atom 继承这个纯虚拟函数而派生出来的，除非提供了函数的主体。这就意味着，如果对象从这些类中实例化的话，则 copy 必须是在从 Atom 直接派生的所有类中来定义；对这种习惯用法的合适的操作是必需的。Atom::copy 没有主体，即隐含着不可能创建一个 Atom 对象；只有导出类才能包含或继承一个带有实例化主体的 copy 成员函数。

让我们看看信件类都做了些什么工作。这些信件类是自主的，并且无须知道信封类的细节。例 5.5 表示了 NumericAtom 类以及此类处理数字原子。构造函数 NumericAtom::NumericAtom(const String &)，从它的 String 参数中“吃”掉一串字符，语法分析为一个整数；第二个构造函数是一个简单的拷贝构造函数，value 函数返回整数值。

例 5.5：NumericAtom 词法的 Atom 的一个类。

```
class NumericAtom :public Atom {
public:
   NumericAtom ():sum(0) { }
   NumericAtom (string &s) {
      sum = 0;
      for (int i = 0; s[i] >= '0' && s[i] <='9'; i ++) {
         sum = (sum *10) + s[i] - '0';
      }
      s = s(i, s.length() -i);
   }
   NumericAtom (const NumericAtom &n) {sum = n.value (); }
```

```
    NumericAtom ()          {sum = 0; }
    long value ()           {return sum; }
    Atom *copy (){
        NumericAtom *retval = new NumericAtom;
        retval ->sum = sum; return retval;
    }
private:
    long sum;
};
```

注意，作为表示数字数量的一个类，NumericAtom 重新定义了 Atom 值的操作，但舍去了字符转换函数 operator char，采用返回空字符的缺省行为。这就意味着，使用这些类的用户必须知道是引用 operator char 这样函数的上下文含义，还是引用 value 的上下文含义。这些函数就不作为纯虚拟的，因为在许多导出类中这些函数不存在重定义的点。copy 成员函数是一种对象的物理拷贝，返回一个 Atom*，这就满足了复制任何 Atom 的单一操作的需要。copy 函数减轻了“类型域”的要求，其描述如下。

例 5.6：类 Name，Punct 和 Oper。

```
class Name :public Atom {
public:
    Name ():n (" ") { }
    Name (String&  s) {
        for (int i = 0; s[i] >='a' && s[i] <= 'z'; I ++) {
            n = n + s(i, 1);
        }
        s = s(i,s.length () - i);
    }
    Name (const Name& m){n = m.name (); }
    ~Name () { }
    String name () {return n; }
    Atom *copy () {Name *retval = new Name;
                    retval ->n = n; return retval; }
private:
    String n;
};
class Punct :public Atom {
public:
    Punct ():c('\0')            {}
    Punct (String& s)           {c = s[0];s = s(1, s.length() -1); }
    Punct (const punct & p)         {c = char(p); }
    operator char () {return c;}
    ~Punct () { }
    Atom *copy() {Punct *retval = new  Punct;
                    retval ->c = c; return retval; }
private:
    char c;
};
class Oper :public Atom {
public:
```

```
    Oper ():c ('\0')  { }
    Oper (String& s)  {c = s[0];s = s(1,s.length() -1); }
    Oper(Oper& o)     {c = char (o); }
    ~Oper () { }
    operator char () {return c; }
    Atom *copy () {Oper *retval = new Oper;
                   retval ->c = c;return retval; }
private:
    char c;
};
```

信件类 Name，Punct 和 Oper 是对 NumericAtom 模拟的逐一实现（见例 5.6）。信封类调用 GeneralAtom，它的工作是从 Atom 的导出类中和谐地创建所有对象：

```
class GeneralAtom :public Atom {
public:
    GeneralAtom(String&);
    GeneralAtom(const GeneralAtom& a) {realAtom = a.copy(); }
    ~GeneralAtom()          {delete realAtom;  }
    Atom *transform ()      {Atom *retval = realAtom; realAtom = 0;
                              return retval;}
public:       // 必须把所有 Atom 的特征进行联合
              // 如果使用 GeneralAtom 就好像它是其导出类的一个实例一样
              // (即，当 transform()不使用时)
    long value ()      {return realAtom ->value(); }
    String name ()     {return realAtom ->name(); }
    operator char ()   {return realAtom ->operator char(); }
private:
    Atom *realAtom;
};
```

作为信封类的 GeneralAtom，只是用户直接处理事务的类：从所有余下类的对象中作为信件处理，存放在信封类的对象内。GeneralAtom 类可以给出一字符串作为构造函数参数(GeneralAtom::GeneralAtom(String&))，并且分析这个串，创建和保留指向合适的 Atom 导出类的对象的指针：

```
GeneralAtom::GeneralAtom(String &s) {
    if (!s.length()) realAtom = 0;
    else switch (s[0]) {
    case '0': case '1': case '2': case '3': case '4':
    case '5: case '6': case '7': case '8':case '9':
        realAtom = new  Numericatom (s);break;
    case ',': case ':': case ';':case '.';
        realAtom = new  Punct (s);break;
    case '*':case '/': case '+':case '-':
        realAtom = new  Oper(s);break;
    default:
        if  (s[0]>= 'a' && s[0] <='z') {
            realAtom = new  Name (s);
        } else {
```

```
            realAtom = 0;
        }
    }
}
```

下面是这个代码的简单引用：

```
Atom *lexAtom(String & s) {
    GeneralAtom g (s);
    return  g.transform ();    // 连续地有效
}
void  parse (String& s) {
    for (Atom *a = lexAtom (s); a; a = lexAtom(s)) {
        ...
    }
}
```

可以用两种方式中的一种来使用信封类实例：一种是作为框架，即在通用上下文中构造一个具体对象的临时对象；另一种是作为实际对象。注意，当 GeneralAtom 被创建时，operator new 的结果对象指针可说明为 Atom*类型；因为从 Atom 派生出 GeneralAtom，并能展示所有相同的行为，所以，这个指针可有效地指向 GeneralAtom。信封类和信件类都是由相同的基类派生出来的，它们是这种习惯用法的重要组成部分；无论是信封类还是信件类，这两个类都蕴含着相同的设计概念。为了替代 GeneralAtom 对象指针的使用，可以用指向 Oper 类、Punct 类、NumericAtom 类等的内部产生的“信件”对象的指针。程序工作在两种情况下都是相同的。在信件情况下，信封函数仅作为框架，以便信件对象在此框架内创建。在这种作为框架使用的情况中，信封不需要最完美的遵守 Atom 导出类的行为。信封的特性不需要反映所表示的所有类型的特性。

在性质上这些信封类对象必须是短暂的，除非它们的特性是所有信封类的特性的联合体。在这个例子中，一个信封类几乎可以用于任何“真实”类所能使用的任何地方，但这是以损害多个间接层作为代价的。这个间接层可以用指向 GeneralAtom 的已存的 Atom*的内容替换来消除，在此指针上有引用 transform 操作的结果，因此此刻信封可以舍去了。

如果信封类的所有操作域确实形成，而这些操作之一要在信封类中被引用的话，那么只引用 transform，以避免多余的间接层，从而改善了性能。这些与多余的间接层共处的打算，可自由地作为自动变量来使用信封类的实例，或者作为全局作用域的初始化对象来使用信封类的实例。信封/信件习惯用法的主要好处是完全隐藏了用户创建对象的动态性质，维护了预期的对象接口。如果不要或不需要透明性的话，则只要把初始化逻辑隐藏到全局函数中，或者隐藏到类的公用静态成员函数中就可以了。

注意，如果构造函数是这样调用的话：

```
GeneralAtom::GeneralAtom(const GeneralAtom&)
```

它就必须得到该对象的拷贝，用此拷贝作为参数来传递。这已经在每个 Atom 导出类中用存放到“类型域”方法实现了，其中 GeneralAtom 将调用合适的构造函数，用合适的办法克隆（clone）该对象。这里我们用虚拟函数 copy 来替代处理。当新的导出类添加的时候，只需对 Atom 或 GeneralAtom 进行改变的是 GeneralAtom 构造函数 GeneralAtom(String&)本身。

虚拟构造函数的另一种方法

本节介绍另一种表达“虚拟构造函数”的方法：当构造函数键入时，创建了不知道精确的类型和大小的对象。这种方法替代了使用带有多余间接层的信封/信件的习惯用法，即在适当的位置上修改对象的类型。此方法建立在伙伴系统（buddy system）内存分配图表上，被称为是伙伴系统虚拟构造函数习惯用法。例如，我们表示这样一个字节流的类，它采用从进入信息链接并把此链接转化为合适的类型对象的字节流，这个类的基础是信息的内容。

这种机制也适用于变长对象。该方法的描述具有某些美学的优点，曾在 Stroustrup 书中介绍过[2]。

当使用这种习惯用法时：当对象大小不知道时，用这种习惯用法最合适。它包括对象类的对象的尺寸不能确定的应用在内（例如，变长信息）。这种习惯用法对性能提高有用，减少了不可忽视的多余数据间接层，另外，在此种情况下指针不能使用（在共享内存中的对象，检查跟踪策略要求预先设计缓冲区等等）。

对于具体数据类型来说，这个习惯用法通常不使用，因为它超出了 C++实现公共使用的假定范围。虽然下面的例子是可移植的，并且未作有关 C++环境实现细节的假设。该习惯用法对大多数高级动态系统资源有用，例如下面的信息缓冲区的例子所作的详细描述。

考虑用于网络应用的基类 LAPD（link access protocol for data，数据链路访问协议）。LAPD 是 ISO 协议的数据链路层（第 2 层）用于公共 HDLC（高层数据链路控制）的子集。在 LAPD 中，我们可以导出另一个高层协议特性的一个类。这里，我们假定有几个不同的信息类型，遵照 X.25 第 3 层（网络层）协议来建立 LAPD 第 2 层协议。这些不同的信息类型表示了不同的通信功能：报警、连接、安装和确认等等。

这个方法的核心是重载 operator new，允许基类构造函数覆盖有导出类对象的对象。当以数据链路字符流作为参数调用基类构造函数时，在动态内存中就建立了此种信息对象。

让我们看一下处理 operator new 的选择重载方案。首先，全局的重载 operator new 安排 LAPD 构造函数调用它的导出类构造函数之一，应当指定在基类对象地址上建立此对象。第二，类 LAPD 有它自己的 operator new，准备在预先确定的缓冲区中确保所建立的对象能够有足够的空间放下最大的信息：

```
// 这种操作符允许在 LAPD 层次结构中一个对象覆盖它自己
// 带有它的导出类之一的一个实例
inline void *operator new(size_t, LAPD *1) {return 1; }
unsigned char bigBuf[4096], *bigBufptr = &bigBuf;
class LAPD {
public:
    void *operator new(size_t) { return (void *) bigBufptr; }
    LAPD (const char *buf) {
        ...
       (void) new (this) Setup(buf);
        ...
    }
    LAPD() {/*无操作*/}
};
```

2 Stroustrup，B.The C++ programming Language．1st ed. Reading，Mass.：Addison-Wesley，1986，p.165.

```
class Setup:public LAPD {
    ...
};
```

此格式的引用为

```
LAPD *msg = new LAPD (buf);
```

作为引用构造函数的准备，调用 LAPD::operator new 为对象分配内存。此对象的内存减少了用 bigBufPtr 指向的缓冲区；然后调用 LAPD 构造函数，将 this 的值设定为 bigBufPtr 初始化点。LAPD 构造函数可以决定用检测 buf 内容的方式调用 Setup 信息，并引用全局重载 operator new，为导出类对象分配内存。在调用 Setup 构造函数之后，全局重载 operator new 就返回到作为参数提供的地址。这样，在 LAPD 构造函数中的 this 的值就被传送到 Setup 构造函数内的 this 值之中。

这种方法所存在的问题是内存需要在调用导出类构造函数之前进行分配，并且编译不知道需要如何预先准备多少内存区。这个问题可以用于在大内存池的底部分配所有信息，并把导出类构造函数的池的底部地址调整到它自己的数据结尾的地方：

```
inline int round (int a, int b) {return ((a+b-1)/b)*b;}
class Setup:public LAPD {
friend LAPD;
private:
    SetupPacketBody body;
    Setup(const char *m) :LAPD() {
        ...
        bigBufPtr += round(sizeof(Setup),4);
    }
};
```

用这种技巧工作总体上不错，但却产生了内存破碎的问题。为了解决内存破碎这一问题，我们使用伙伴系统内存管理算法的变型。本章的代码都是以此种说明格式为基础而编制的。

在伙伴系统中，要求给定尺寸的块要满足其他最大的 2 次幂的尺寸块。未用过的块由空闲表来维护。如果最近的空闲块靠近前一个空闲块的话，那么这两个块就接合成一个，使内存破碎减少到最小。伙伴系统方法的有趣的特性是邻近块的地址能够由最近的空闲块地址和尺寸中推断出来，使之可以容易决定哪一块能够被接合。

替代寻找最近的合适块的方法是用伙伴系统去做“最差的合适”工作，即用最大的合适块去满足信息缓冲区的要求。但是，分配算法在标识这个块之后挂起，并在完成分配处理之前允许“虚拟”构造函数继续执行。当完成信息的语法分析时，对象尺寸就已知，此时将控制返回到伙伴系统分配符，将大块进行分割，所以封闭的内存块是其他最大的 2^k 块的封闭对象。

现在，当此块空闲时，它就享有伙伴系统将邻近块接合成单一的较大块的好处，内存破碎程度最大限度地减小了，提供的内存池比最大信息的尺寸还要大得多，并且维护了“虚拟构造函数”完全的灵活性。

例 5.7 给出了 LAPD 基类的接口，展示了 new 和 delete 操作符的接口，作为以 const char*为参数的构造函数，应用程序使用所建立的信息，提供缺省的构造函数；这将从导出类构造函数中来引用。因为是引用前一个初始化对象，所以必须作为无操作来定义。调用两次这种基类对象的构造函数是值得的——在用户直接层上调用 LAPD(const char*)，并从导出类调用 LAPD()——所以，任何

LAPD 的类对象成员用第一个构造函数来设定，用第二个构造函数缺省值来重新初始化。换言之，这个基类不应当包含类对象实例。

例 5.7：LAPD 类。

```
class LAPD {
friend class LAPDMemoryManager;
public:
   virtual int size() { return 0; }
   void *operator new (size_t);
   void operator delete(void*);
   LAPD (const char *const);
protected:
   LAPD () {/*无操作*/}
private:
   // 因为在伙伴系统中，每个结点既可以是一个空闭块，又
   // 可以是分配的内存块，所以分配的块不需要链接表的
   // 内存消息的开销
   union {
      struct {
         unsigned char flag;
         struct {
            unsigned int sapi:6;
            unsigned int commandResponse:1;
            unsigned int zero:1;
            unsigned int tei:7;
            unsigned int ext:1;
         } address;
         unsigned char control;
      } header;
      struct {
         LAPD *linkf, *linkb;
         unsigned short size;
      } minfo;
   };
   // 无论是分配的块,还是空闲表块都
   // 需要垃圾收集这个位(6:+)
   unsigned char tag;          // 伙伴系统分配标记
   int performCRCCheck () {/*...*/}
};
```

类的大多数内部数据都专门存储在 LAPD 报文头的域中，并为动态内存管理定义域。仅在对象已经返回到空闲表之后，内存管理的许多域方可使用，它们的空间使用联合（union）与应用实例空闲共享。域 tag 必须引入到这样的对象中，是在应用中激活，还是在空闲表上激活，要由 tag(标记)域来决定。这个 tag 促使伙伴系统垃圾收集器查看并确定是否与分配内存相关的块有关。

例 5.8 表示了一个类用于管理内存消息缓冲区的情况。伙伴系统有它自己的隐藏在应用程序中的数据结构，并且这些结构需要在应用程序分配消息缓冲区内存之前就初始化。所以，内存分配实现就放进它自己的类中，以代替遍及消息类本身的分配。一个单一的、全局的 LAPDMemoryManager 的实例，仅调用 manager，为所有 LAPD 消息的内存管理服务。

例 5.8：消息内存管理器。

```
inline int round (int a, int b) {return ((a+b-1)/b)*b; }

// 这个类能够嵌套在 LAPD 中
// 它建立了用于管理消息内存的单个的惟一的对象
// 建立这个类的主要原因是在伙伴数据结构中设定
// 初始化条件
class LAPDMemoryManager {
friend LAPD;
public:
   LAPDMemoryManager ();
   LAPD *largestBlock();
   void allocateResizeBlock(int);
   void deallocateBlcok(LAPD, int);
private:
   enum { MessageBufSize=4096, Log2MessageBufSize = 12 };
   unsigned char availBuf {
      (1 -Log2MessageBufSize) *round(sizeof(LAPD),sizeof(LAPD*))};
   // avail 包括这样一些对象，它服务于不同尺寸对象空闲查询的列表头，总共为 2**k
   LAPD *avail;
   int savej;          // 存入的上下文遍及了最大块(1argestBlock)和
                       // 分配尺寸块(allocateResizedBlock)的调用
   unsigned char buf[MessageBufSize};
   LAPD *buddy (int k, LAPD *1) {
      char *cp = (char *)1;
      return (LAPD*) (1ong (cp) ^(1 << (k+1)));
   }
};
static LAPDMemoryManager manager;      // 单个、惟一的对象
```

内存管理类的接口包含一个缺省构造函数，它只初始化了伙伴系统的数据结构：

```
LAPDMemoryManager::LAPDMemoryManager () {
   // 数据结构初始化
   LAPD *buf=(LAPD *)this ->buf;
   avail = (LAPD *)availBuf;
   avail [Log2MessageBufSize]. minfo. link f = 0;
   avail [Log2MessageBufSize]. minifo. linkb = 0;
   buf[0].minfo.1inkf = buf[0].minfo.linkb =
                       & avail [Log2MessageBufSize];
   buf[0].tag=1;
   buf[0].minfo.size=Log2MessageBufSize;
   for (int k = 0;k<Log2MessageBufSize;k ++) {
      avail [k].minfo.linkf = avail[k].minfo. linkb = & avail [k];
   }
}
```

伙伴系统分配算法划分成两个成员函数并进行修改，其目的是使对象能在传送第一阶段的分配之后继续生长。largestBlock 成员函数返回一个指向在伙伴系统空闲池中最大可用块的指针：

```
LAPD *
```

```
LAPDMemory Manager::largestBlock () {
    for (int k = Log2MessageBufsize; k >=0;--k) {
        if (avail [k].minfo.linkf ! = &avail [k]) {
            savej = k;
            return avail[k].minfo.linkf;
        }
    }
    return 0;
}
```

私有向量 avail 包含的对象，对所有二次幂的不同尺寸的对象空闲池表作表头（报文表头）服务。用正规的伙伴算法查找这个向量，以寻找至少是像所期望的空间一样大的最小尺寸的块，然后分别将要求最合适的大块空间划分成两个相等的块。因为 LAPD 类在开头时不知道需要多少空间，所以要修改该算法以返回最大的可用块，在内存管理和 LAPD 导出类之间安排一半的空间大小。

deallocateBlock 成员函数返回空闲存储区一个块，如果这种情况存在的话。就与邻近的空闲块接合在一起。allocateResizeBlock 成员函数实现剩余的伙伴系统算法。例 5.9 是这种算法的代码。

例 5.9：LAPD 类的块分配子程序。

```
void
LAPDMemoryManager::allocateResizeBlock(int sizeNeeded) {
    int k, j = savej;
    // 只入不舍到下一个 2**n
    for (int i =1; i < Log2MessageBufSize; i ++) {
        k = 1 << i;
        if (k > sizeNeeded) break;
    }
    LAPD *1 = avail [j]. minfo. linkf;
    avail [j]. minfo. linkf = 1 ->minfo.linkf;
    1 -> minfo.linkf ->minfo.linkb = &avail [j];
    for (1 ->tag = 0;j - k;) {
        --j;
        LAPD *p = (LAPD*) (((char *)1) + (1 << j));
        p ->tag = 1;
        p ->minfo.size = j;
        p ->minfo.linkf = &avail [j];
        p ->minfo.linkb = &avail [j];
        avail [j].minfo.linkf = p;
        avail [j].minfo.linkb = p;
    }
}

void
LAPDMemoryManager::deallocateBlock(LAPD *1, int k) {
    for (;;) {
        LAPD *p = buddy (k,1);
        if (k == Log2MessageBufSize || p ->tag == 0 ||
                                p ->minfo.size != k) {
            break;
        }
```

```
        p ->minfo.linkb ->minfo.linkf = p ->minfo.linkf;
        p ->minfo.linkf ->minfo.linkb = p ->minfo.linkb;
        ++k;
        if (p < 1) 1 = p;
    }
    1 ->tag = 1;
    1 ->minfo.linkf = avail [k].minfo.linkf;
    avail [k].minfo.linkb = 1;
    1 ->minfo.size = k;
    1 ->minfo.linkb = &avail [k];
    avail [k].minfo.linkf = 1;
}
```

在 LAPDMemoryManager 的私有实现中，我们发现 savaj 成员用于把索引的值保存到交叉的两段分配算法 avail 中。该索引能够确定正在工作块的大小和位置。我们也找到缓冲区大小用于分配算法的规格说明以及二进制日志的值，buf 缓冲区本身被封装在内存管理对象中。空闲表的报文头向量被称为 availBuf，说明为字节的匿名向量但要通过指针 avail 访问，就好像是 LAPD 对象向量一样。最后，我们找到 buddy（伙伴）函数的拷贝，用于标识可能接合到一起的邻接块。buddy 代码似乎不可能移植：char*指针抛弃了执行异或操作的 long，支持伙伴系统设计方案。如要进行调整可能所需要的指针比用 long 还要复杂的机制。

在此需要拟定消息内存管理的基本成分，这个软件可驻留在对用户隐藏的库中，下面我们来讨论一个例外。应用程序员用 LAPD 派生的方式创建消息类，例 5.10 的 Setup 和 Conn 类就是这样做的。除了调用 allocateResizeBlock 以外，代码直接在构造函数内，从伙伴算法的第二阶段开始调整消息的实际尺寸块的大小。

例 5.10：调用 Setup 和连接第 3 层协议类。

```
class Setup:public LAPD {
friend  LAPD;
private:
    struct SetupPacketBody {
        unsigned char rep [4096];
    };
    SetupPacketBody  body;
    int size () { return sizeof (Setup); }
    Setup (const char *m) : LAPD () {
        manager.allocateResizeBlock(size () );
        ::memcpy(&body, m + sizeof(header), sizeof(body));
        ...
    }
};
class Conn:public LAPD {
friend  LAPD;
private:
    Struct ConnPacketBody {
        unsigned char rep[4096];
    };
    ConnPacketBody body;
    int size () {return sizeof (Conn); }
```

```
    Conn (const char *m):LAPD () {
        manager.allocateResizeBlock (size() );
        ::memcpy (&body,m + sizeof(header), sizeof(body));
    }
};
```

注意，在消息类的设计中，我们要把消息的主体和网络上与相同数据一致的抽象（包括报文头位和报文尾位等）之间加以区别。这里，前者调用 SetupPacketBody，而后者调用 Setup。例 5.10 表示了处理 Conn（连接）信息的类的类似的结构。

现在我们来表示一个消息是如何建立的。一个应用程序从输入数据链路中获取字节流，并把此字节流作为参数传送到 LAPD 构造函数中（见例 5.10）。LAPD::LAPD(const char*)需要确定消息类型，所以可以引用合适的导出类构造函数，引用形式类似于：

```
LAPD *message = new LAPD(bytepointer);
```

这种引用将调用 operator new，为对象分配存储区。一个特别的调用就是对 LAPD 类的定义，而且所有处理的是请求最大块的地址的内存管理并返回其值：

```
void*LAPD::operator new(size_t /*未使用 */)
    return manager.largestBlock();
    // 在导出类构造函数中获得完成
```

分配现在完成了，LAPD 构造函数能够确定消息类型和调用合适的导出类构造函数。例如，假定消息的第 4 个字节（报文头后的数据）用于确定消息类型（见例 5.11）。

例 5.11：建立消息对象的构造函数主体。

```
LAPD:: LAPD (const char *bits) {
    ::memcpy (&header, bits, sizeof(header));
    performCRCCheck();
    // 确定消息的类型是基于消息内容域的第四个八比特组（字节）
    switch ((bits + sizeof (header)) [3]) {
    case SETUP:
        (void) ::new(this) Setup (bits); break;
    case CONN:
        (void)::new(this) Conn(bits); break;
    case ALERI:
        (void) ::new(this) Alert (bits); break;
    ...
    default;
        error("invalid message"); break;
    }
}
```

在识别出消息类型的基础上，LAPD 构造函数就能直接调用合适的导出类构造函数，语法为

```
(void) new(this) derived_class (arg)
```

调用产生了重载 new 的定义：

```
// 这种操作符允许在 LAPD 层次结构中一个对象覆盖它自己带有它的导出类之一的一个实例
inline void *operator new(size_t, LAPD *1) {return 1; }
```

换句话说，导出类在基类实例驻留的相同位置上强制分配其实例，所以正好覆盖了已存的实例。记住，这是在内存分配器最大可用块的底部。

如果要补充细化消息类型的话，从 LAPD 导出的类可能对它的导出类之一涉及更详细的消息处理过程。例如，Setup::Setup 可以覆盖带有它自己导出类之一的实例的当前 Setup 实例。最后细化过程停止，而导出类构造函数将调用内存管理的 allocateResizeBlock 成员函数，以便最后定下块的大小和重新组织空闲内存表。

内存的撤销更是直截了当。每个消息都调用 LAPD::operator delete，以释放它所占有的存储区。进而就调用内存管理的 deallocateBlock 成员函数，让伙伴系统收回此块：

```
void
LAPD::operator delete (void *1) {
    manager.deallocateBlock((LAPD *)1, ((LAPD*)1) ->size());
}
```

可变长对象

除了沿继承方向的某些类涉及到匹配对象的尺寸以外，上述例子能够扩展，创建给定类的可变长对象。可变长对象在给定类型的消息有不同主体的尺寸的消息应用中有一定的价值。

由 Stroustrup 推荐的原始方法已不再使用，因为对 this 的赋值在 C++ 2.0 和 C++ 2.0 以上的版本中是非法的。在 C++中引入 operator new 的一个原因是提供了更通用的问题解。一个类可以对它的所有实例安排具体的尺寸：

```
class LAPD {
public:
    // 使所有 LAPD 对象有 128 字节大
    void *operator new(size_t) {return::new[128]; }
    ...
};
```

但是，这个方法还有几个缺点。首先，operator new 不访问构造函数参数，所以对象尺寸参数仅通过全局数据传递。第二，构造函数执行得太晚，以至于对对象尺寸的确定几乎成为不可能：在构造函数主体执行之前，operator new 就完成了。第三，在任何基类中，无显式方法调整对象尺寸。

用前面所描述的伙伴系统算法，我们可解决上述这些问题。正如原始的 LAPD 类一样，只有对象初始化最后完成了，对象尺寸才保留下来。内存管理器通知该对象分配多少个 total 字节，所以对象的尺寸可以扩展，但却受到该类尺寸的限制。在例 5.12 中，类 Conn 有预先计算的值 bodySize（包含了信息主体的尺寸），这个尺寸可以计算出整个对象的大小。主体被拷贝到 bodyStorage 中，其行为就像可变长向量一样，在构造函数中用 memcpy 调用。

例 5.12：动态长度的 Conn 类。

```
class Conn:public LAPD {
friend LAPD;
private:
    struct ConnPacketBody {
        unsigned char rep[4096];
    } *body;
    int bodySize;
```

```
    int size() {return sizeof (Conn) + bodySize; }
    Conn (const char *m):LAPD () {
        ...
        ::memcpy (&bodyStorage, m + sizeof(header), bodySize);
        body = (ConnPacketBody*) &bodyStorage;
        manager.allocateResizeBlock(size() );
    }
    char bodyStorage[1];
};
```

在对象建立之后，内存分配器就告知对象的固定长度为 size()，其中包括该消息主体的存储区。

注意，使用这种技术的类比采用具体数据类型要有更多的限制。这些类不可以创建能够产生任何上下文的原始对象，这些类只能在预分配的内存块中实例化，其地址是由它们的构造函数来传递的。

协同委托和信封类

继承的动态、运行时模拟能够做类似于委托的工作。委托是由对象实例继承类的部分特性所构成：其能力是共享代码并自动地使用由另一域组成的抽象代码。委托是基于一个对象成员函数，并由另一个对象提供的上下文之间的运行时关系所决定。程序设计语言 Actors 和 Self 都支持委托。

在 C++中，继承关系与编译-时间耦合在一起。C++不直接支持委托——它的类型系统必须在运行时加以深入研究——但委托的许多方面可以通过一个对象成员函数调用另一个转发来模拟。这种转发可以在类的范畴内执行，对继承的使用是相同的。一种可能要模拟继承的是使用成员函数转发到有助于实现子类型的行为，以及运行时符号联编的灵活性。但是，管理工作放在“焦点特写”上了，这样会带来令人乏味的局面：

```
class A {
public:
    void a ();     // 限于 A
    void b ();     // 限于 A
    void c ();     // 限于 A
    A(char *r) {
        cp = new char [::strlen(r) + 1]; ::strcpy(cp,r);
    }
private:
    char *cp;
};
class B {
public:
    void a ()      {delegate ->a(); }    // 乏味的
    void b ()      {delegate ->b(); }    // 乏味的
    void c ()      {delegate ->c(); ]    // 乏味的
    void d ();         // 仅限于 B 类
    B(char *r = 0)     {delegate = new A (r); }
private:
    A *delegate;
};
```

如果我们使用继承来实现子类型的话，继承成员函数就出现了空闲。这里，在类 B 中复制 A 的接口

对初始代码和演变都是令人乏味的：任何接口改变必须在两个类之间对等。

复制接口的许多不便之处在 C++中可以用重载操作符->取代对外部对象的间接引用指针的语义而加以缓和。实现技术与使用计数指针技术相同，它允许一个对象透明地把操作“转发”到另外一个：

```
class A {
public:
   void a ();          // 限于A
   void b ();          // 限于A
   void c ();          // 限于A
   A(char *r) {
      cp = new char [::Strlen(r)+1]; ::strcpy (cp, r);
   }
private:
   char *cp;
};
class B{
public:
   // a, b, c 不必要的显式转发
   void d();      // 限于B
   A *operator ->()  {return delegate; }
   B (char *c = 0)   {delegate = new A(c); }
private:
   A *delegate;
};
```

对象委托能够在任何时候通过采用改变 delegate 指针值的方法来加以改变。

现在，操作能够自由地添加到类 A 中，无须担心 B 中存在改变是否对等的问题。有 4 个其他因素需要加以注意：

（1）除了它的构造函数以外，如果 A 改变的话，B 的代码不一定要重编译。

（2）这些对象的说明和使用的语法采用如下形式：

```
int main() {
   B aBInstance("hello world\n");
   aBInstance.d();          // 用(.)点表示(很少使用)
   aBInstance ->a();        // (->)箭头表示
                            // 没有删除 aBInstance:它不是一个指针
   return 0;
}
```

这个方法对用户是一个沉重的负担，因为要记住类的一个成员函数是转发了还是没有转发，并且还要使用合适的语法。如果这是一件相当麻烦的事情，那么操作符 operator->不应当使用，并且每个成员函数应当一个一个地、单独地像处理以上的“令人乏味”的例子那样来进行处理。如果所有函数都被转发了，那么当然所有成员函数调用的语法就是一致的。

如果我们能够重载 operator.的话，那么这些不方便之处就会消失，但 C++的少数操作符可能不会重载。

（3）在类的内部，我们必须使用箭头(->)而不使用点(.)来表示引用“真实”的成员函数。点(.)

表示仅对函数使用，不对委托使用，但要在类的外部来定义。

（4）如果操作符（如算术操作符）像 A 的类成员一样被重载的话，那么它们也必须在 B 中定义：

```
int A::operator + (const A &a) const {
   int q;
   ...
   return q;
}
int B::operator + (const A &a) const {
   return delegate ->operator + (a);
}
```

例 5.13 是另一个例子。这里，类 SeaPlane 在功能上既可以委托 Boat 又可以委托 Plane，这依赖于它的“方式”。

例 5.13：SeaPlane 类模型的委托。

```
class Vehicle {
public:
   virtual int mpg ()          { return 0; }
   virtual void takeOff ()     { hostVehicle ->takeOff(); }
   virtual void swim()         { hostVehicle ->swim(); }
   vehicle (Vehicle *host)     { hostVehicle = host; }
private:
   Vehicle *hostVehicle;
};
class Plane: public Vehicle {
public:
   int mpg() { return 10; }
   Plane (Vehicle *v) : Vehicle (v) { }
};
class Boat : public Vehicle {
public:
   int mpg() { return3; }
   Boat (Vehicle *v) : vehicle (v) { }
};
class SeaPlane : public Vehicle {
public:
   SeaPlane () : Vehicle (0) {
      currentMode = boat = new Boat (this);
      Plane = new Plane (this);
   }
   ~SeaPlane () { delete boat; delete plane; }
   void takeOff() { currentMode = plane; }
   void swim () {currentMode = boat; }
   Vehicle *operator ->() {return currentMode; }
private:
   Vehicle *boat, *plane, *currentMode;
};
int main() {
   SeaPlane seaPlane;
```

```
    cout << seaPlane ->mpg() << endl;
    seaPlane ->takeOff(); cout << seaPlane ->mpg() << endl;
    seaPlane ->swim(); cout << seaPlane ->mpg() << endl;
}
```

当使用这种习惯用法时：在信封/信件类对中使用这种习惯用法，为了避免信封和信件类接口特性的更新冗余，作一些改变是经常的。这种习惯用法也是其他高级习惯用法的基础，诸如后面第9章的符号习惯用法。

迭代器和游标

收集类（collection class）（诸如集合、列表和树）在C++程序中是公用的实用程序。收集类可以在几个类之间共享。例如，在模拟程序中的资源列表可以由其生产者、消费者和模拟控制类共享。所有这些类都可能在此列表上连续地访问每个元素，它们有着不同的理由：生产者可能会改变先前队列项的状态；消费者寻找具有某种优先级和特征的项；模拟控制者打印出所要求的队列的内容。

所有这些搜索是无损的搜索：在它们自身中并不影响列表的状态。如果这些搜索是隔行扫描的，那么其关键是不修改列表内部数据。但却需要连续地访问每个列表成员的数据，对列表本身来说是私有的。几个对象（生产者、消费者和控制者）如果能保持“列表位置”状态的自己的拷贝的话，就可以独立地遍历每个列表。这种状态在对象中的保持就称为迭代器或游标（指针），即让占有者草草地浏览，每次只收集一个元素。

迭代器是处理类的特殊种类，与以前章节所述的处理类不一样，迭代器是补充的辅助类，但不替换主体类接口。通常情况下，迭代器在它们的主体的friend中加以说明。

考虑在本章结束时表示的Queue类，当编译和运行时，这个程序产生如下输出：

```
1 2 3
leaving: 1 2 3 4 5
6 7 8 9 10 11 12
leaving: 6 7
8 9 10 11 12
leaving: 8 9
```

Queue迭代器的作用就像“游标”一样，草草地浏览，一次只收集一个元素。Queue对象的每个游标实例化是独立于其他游标的，并且迭代器的使用不影响Queue本身。

即使，Queue在迭代器使用之间有所变化，这个代码也必须运作。任何一个Queue状态都不能用QueueIterator交叉引用它的成员函数而被牢记。

这里，迭代器从const引用到Queue的初始化，在试图遍历之前，确保Queue的存在。

5.6 功能元件：作为对象的函数

在2.9节中，我们看到类成员函数指针是如何通过相同的接口提供几个不同的函数版本的。但是，按照这种方式调用一个成员函数“参数化”的语法使用起来却不是很方便：

```
...afilter. *(afilter. current)(t)...
```

并且该方法的大多数协调工作是基于C++语言的模糊结构的。使用虚拟函数的继承，我们能够将对这种情况的处理变得清晰些。

回忆 2.9 节所讨论的当前开发的应用例子，这个例子的电路网络是由电感、电容和电阻组成，这些分量依赖于它们的参数值，电路的作用选择这三种方式中的一种，每种响应分别使用各自的函数。

在这里，我们建议每种不同的响应类型在它自己的类中编码，以替代它自己的函数。我们把这种高级组织结构称为功能元件（functor）。逻辑上，功能元件是和对象行为一样的函数（或者像函数行为那样的对象）：功能元件是为函数角色服务的，但它可以被创建，可以作为参数传递，并且可以像对象那样被操作。在 C++中实现是直接的：每个功能元件是有单一成员函数的类。功能元件对象能够表示电路的 3 种可能响应手段的各自特征。

当使用这种习惯用法时：每当使用这种习惯用法时，就需要使用函数指针，特别是成员函数指针。

基于以上论述，我们应当做些什么呢？首先，使用继承，我们能把公用因子代码放进基类之中。这些公式包含了一些子表达式，在它们自己的权限中估计可能作为函数是足够复杂的。在这个特殊的例子中，子表达式（如$\omega 0$ 出现在几个公式中）在应用领域中与定义好的构造函数有关。这些固有的系数作为公用部分加以处理。此外，这样使用的继承提供了一个与函数联系在一起的语言结构。所有功能元件有一个公共的基类，所有的功能元件在给定的上下文中能够互换。

第二，使用带继承的虚拟函数来实现函数指针，我们能获得同类的运行时灵活性。替代选择一个函数指针，就像引用一个“真实”函数特征值一样，我们指定一个指向合适功能元件对象的指针。我们调用函数就像此函数驻留在基类一样，并且在运行时“间接”地选择正确函数，用 C++的虚拟函数机制来加以处理。

这里存在着不同的实现问题。我们可以很容易地设计相应于强阻尼、弱阻尼和临界阻尼响应的 3 个类，我们能使基类包含所有这 3 个类。然后再虚构一个机制以决定建立 3 个功能元件中的哪一个。在该函数指针的例子中，我们在现有的构造函数内含有所有的数据，并且能够使用这些数据来决定选择哪一个函数。这里，带有功能元件的行为本身应该用创建的对象来标识。到任何构造函数都在运行的时候才决定哪一个对象的创建已为时太晚。键入信封/信件习惯用法，模拟虚拟构造函数，提供额外的间接层来支持动态类型对象的创建。

例 5.14：基功能元件类和 3 个模拟方式的类。

```
#include <complex.h>
typedef double time;
class SeriesRLCStepResponse {
public:
   virtual complex current(time t) {
      return object ->current (t);
   }
   double frequency () const {return 1.0/sqrt (L * C); }
   SeriesRLCStepResponse(double r, double i,
      double c, double initialCurrent, short isenvelope = 1);
protected:
   double R, L, C, currentT0;
   double alpha;
   complex omegad, a1, b1, a2, b2, s1, s2;
private:
```

```
    SeriesRLCStepResponse *object;
};
class UnderDampedResponse: public SeriesRLCStepResponse {
public:
    UnderDampedResponse(double r, double l, double c, double i):
        SeriesRLCStepResponse(r, l, c, i, 0) { }
    complex current (time t) {
        return exp (-alpha *t) *(b1 *cos (omegad *t) +
            b2 *sin (omegad *t));
    }
};
class OverDampedResponse:public SeriesRLCStepResponse {
public:
    OverDampedResponse (double r, double l, double c, double i):
        SeriesRLCStepResponse (r, l, c, i, 0) { }
    complex current (time t) {
        return a1 *exp(s1 *t) + a2 * exp (s2 *t);
    }
};
class CriticallyDampedResponse: public SeriesRLCStePResponse {
public:
    CriticallyDampedResponse(double r, double l, double c, double i):
        SeriesRLCStepResponse(r, l, c, i, 0){ }
    complex current(time t) {
        return exp(-alpha *t) * (a1 *t + a2);
    }
}
```

例 5.14 表示了用于谐振电路模型的类。SeriesRLCStepResponse 既可作为每个信件的基类，又可为信封本身服务。对信件类的基类来说，它提供了一个虚拟成员函数 current，可用导出类逐一地重新定义。基类对其所有导出类的公用特性都含有 protected 数据。

对一个信封类来说，它有一个命名为 object 的指针，指向该信件，处理各个单一的响应类型。这个域是 private，所以它不输出到导出类。信封把 current 成员函数的调用转发到信件的虚拟函数 current，其行为将在运行时装订。

例 5.14 描绘了电路操作的 3 种选择的每个功能元件类。对于给定的电路模拟，要实例化这些类的一个单一对象。每个类有两个公共(public)成员函数：它的构造函数，需要收集当前赋值的参数，而 current 产生一个时间函数的当前值。在基类 SeriesRLCStepResponse 中，虚拟说明 current 函数。

现在，我们返回基类构造函数，其中所执行的大多是有趣的工作（见例 5.15）。构造函数给确定的电路操作模式的自变量赋值，然后把基类 object 域装订到最近建立的 SeriesRLCStepResponse 的合适的导出类的“响应”对象上。

例 5.15：电路响应类的构造函数。

```
SeriesRLCStepResponse::SeriesRLCStepResponse(
        double r,double l,double c,double initialCurrent,
        short isenvelope) {
    R = r; L = l; C = c; currentT0 = initialCurrent;
    alpha = R / (L + L);
```

```
        ...
        // 计算 a1,b1,a2,b2 等
        ...
    if (isenvelope) {
        if (alpha < frequency ()) {
            object = new UnderDampedResponse(
                                        r,l,c,initialCurrent);
        } else if (alpha > frequency ()) {
            object = new OverDampedResponse(
                                        r,l,c,initialCurrent);
        } else {
            object = new CriticallyDampedResponse (
                                        r,l,c,initialCurrent);
            }
        } else {
                omegad = sqrt (1.0/(L*c) - (alpha*alpha));
        }
    }
```

注意，这个构造函数采用以下两种方式之一来运作：它既能作为信封调用产生的对象，又能调用基类的一部分的导出类。如果把调用 isenvelope 参数设定为 1，那么构造函数的行为就好像对象是在它自己权限之内的应用对象。该对象作为信封来分析电路参数并建立信件对象模式的电路行为。如果 isenvelope 是 0，那么该构造函数就假定调用导出类中的一个。在这种情况下，惟一的工作是计算ω_d。isenvelope 值，缺省时是 1；导出类必须重写缺省的参数规格说明，并传送到基类的构造函数。

主程序是简单明了的：

```
int main () {
    double R,L,C,I0;
    cin >> R >> L >> C >> I0;
    SeriesRLCStepResponse afilter (R,L,C,I0);
    for(time t = 1.0; t < 100; t += 1.0) {
        cout << afilter.current(t) <<endl;
    }
    return 0;
}
```

该程序建立了带有合理参数的 SeriesRLCStepResponse 对象，然后分别调用带有不同时间数值的 current 操作。从用户观点来看，使用的简易性与 2.9 节的实现如出一辙。但是，这个实现更加简单，并且展示了更为丰富的美学设计。提供方便使用的大多数的函数指针均可用 C++虚拟函数来实现，产生的代码也很容易理解、诊断和维护。

函数的和应用的程序设计

以上方法提供了功能元件的较好的概念模式，但函数仍然不像我们所要的那种对象。使用 C++重载，我们能在句法层和完美的功能元件习惯用法上把函数和对象紧密地结合在一起。

除了提供 current 函数之外，我们还使用 operator()（“调用操作符”）来提供对相同函数的访问：

```
class SeriesRLCStepResponse {
```

```
public:
   virtual complex operator () (time t) {
      return current (t);
   }
   Virtual complex current (time t) {
      return Object ->current(t);
   }
   double frequency () const {return1.0/(L * C); }
   SeriesRLCStepResponse (double, double,
                          double, double, short = 1);
   ...
};
```

现在我们可以使用滤波器(filter)，所建立的一个对象就好像一个函数一样：

```
SeriesRLCStepResponse afilter(R, L, C, I0);
for(time t = 1.0; t < 100; t += 1.0) {
   cout << afilter (t) << endl;
}
```

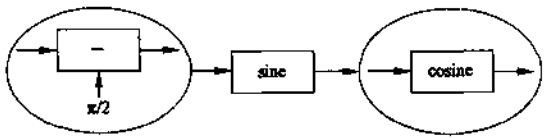
这可能是最容易接受的句法，但它有着某些强有力的应用。考虑所应用的程序设计领域，其中函数是基本的抽象：函数就像值一样处理。主要是用一个函数外加到此抽象上，通常的结果是产生另一个函数。一个函数应用的结果或者是一个数据值，或者是另一个函数！

这里，我们讨论一下对电路滤波器模拟的功能元件的应用。滤波器是一种电子过滤装置，这种装置允许特定频率范围的信号通过，过滤掉其他频率信号。例如，若电视频道包含了带通滤波器（band-pass filter），在电视机屏面上能够观看到各个频道所选取的信号，拒收其他频率的信号。你也可以在天线上放置一个“槽口”过滤器，使附近的收音发射机的频率不能干扰到电视画面。我们将使用功能元件对象建立这样的滤波器的一些模型，每个对象用按规格改制的滤波器响应的特性来初始化。功能元件对象引起输入、传送消息并产生输出结果。

功能元件的滤波器模型，我们认为作为数学抽象是有用的，也可以认为电子滤波器是数学函数的实现：输入值连续地映像到输出值。如果我们把一个输入外加到滤波器上，经过滤波器的转换就产生一个输出，这个过程类似于一个数学函数。我们表示的是有形抽象的函数，一个对象，正如现实世界的电子滤波器是有形实体的一样。

我们也把电压值作为函数来建模。对这里的应用来说，电压是数量和频率的常数，但可以想像成它们是动态的。

在应用的范例中，把一个输入值外加一个函数（即函数值）中就产生一个结果。把一个电压加到一个滤波器中就会产生另一个电压。但是，把一个函数外加到另一个函数又是什么样的结果呢，或者说在我们这种情况中，把一个滤波器外加到另一个滤波器会是什么样的结果呢？考虑一个 sine 函数减去 $\pi/2$ 的代数应用，它是独立于任何数字量的应用。替代这个值的结果是 cosine 函数本身。



当函数是一个值时，它们可以使用另一个函数操作，并产生其他函数。我们也可以用同样的方

式处理滤波器。当然，归根结底，对滤波器外加一个电压值将会产生一个电压和频率的输出值。

例 5.16 表示通用应用值类，它既用于表示电压值，又用于表示电子滤波器类的基类。通用应用值类输出一个类型 Type，即每个滤波器类用于决定滤波器装置的类型，所以它可从其本身和驱动滤波器中综合新的滤波器实例。

例 5.16：应用通用电子信号值。

```
typedef double Frequency;
typedef double Voltage;
class Value{
public:
   enum Type {T_LPF, T_HPF, T_BPF, T_Notch, T_Data };
   virtual Type type () {return T_Data; }
   virtual Frequency f1 () (return omega1; }
   virtual void print () {
      cout << "Value", << volts << "volts at frequency"<<
         omega1 << endl;
   }
   virtual Value *evaluate (Value* = 0) {
      return this;
   }
   Value(Voltage v=0, Frequency w1=0):volts(v), omega1(w1){
   }
private:
   Frequency omega1;
   Voltage volts;
};
Value Zero(0);
Value *zero = &Zero;
```

每个滤波器通过 type 成员函数发布它的类型。所有值和所有滤波器至少有一个频率；对于值来说，这是它们的特性频率。而对于滤波器来说，它指定一个截止点。print 成员函数表示通用打印结果。最后，如果已经用输入信号来驱动的话，我们提供一个 evaluate 成员函数，并产生一个值或滤波器，返回输出电压。

例 5.17 表示了 Filter 抽象，此抽象重写了基类 Value 的所有成员函数。一些函数（evaluate，operator()和 print）是用类 Filter 构成的纯虚拟函数，促使它的导出类提供自己的版本。添加一个成员函数 f_2，使滤波器切断两个频率点而用一个来代替，并用私有数据项 omega2 存储其值。类 Filter 也添加了一个纯成员函数重载调用函数 operator() (Value *)，这个操作符允许对象“被调用”，就好像它是一个函数一样。如果应用需要，多重重载 operator()可以提供不同参数的列表。

例 5.17：通用电子滤波器抽象类。

```
class Filter:public Value {
public:
   Filter(Frequency w1, Frequency w2 = 0):
      Value(0, w1), omega2(w2), cachedInput (0) {
   }
   virtual Value *evaluate(Value* = 0) = 0;
   virtual Value *operator () (Value*) = 0;
```

```
    virtual void print () = 0;
    Frequency f2 () {return omega2;}
    Type type () {
        if (cachedInput) return cachedInput ->type();
        else return baseType;
    }
protected:
    Type myType, baseType;
    Value *cachedInput;
private:
    Frequency omega2;
};
```

我们可以由 Filter 建立低通滤波器（low-pass filter，简称 LPF）的模型来装配一个新类：

```
class LPF:public Filter {
public:
    Value *evaluate(Value*input = 0) {
        Value *f = cachedInput ->evaluate(input);
        if (f ->f1() < f1()) return f;
        else return zero;
    }
    Value *operator () (Value*);
    void print () {
        if (cachedInput) cout << "COMPOUND" << endl;
        else cout << "LPF(" << f1() << ")" <<endl;
    }
    LPF(Frequency w1):Filter(w1), baseType(T_LPF) { }
};
```

它的 print 操作和构造函数是直接的。Evaluate 成员函数传播了滤波器（或值）连接它的输入的求值要求，该传播直到达到电压值对象为止，并且最后返回一些信号值。求得信号的频率似乎是落到滤波器带通之内（即是否该信号应该通过或者被吸收），并返回一个合适的值。

更有趣的是，为这个类定义了 operator()(Value*)：

```
Value*
LPF::operator () (Value*f) {
    switch (f ->type()) {
    case T_Data:
        myType = T_Data;
        cachedInput = f;
        return evaluate ();
    case T_LPF:
        if (f ->f1() > f1()) return this;
        else return f;
    case T_HPF:
        if (f ->f1() > f1())return new Notch(f1(), f ->f1());
        else return new BPF(f ->f1(), f1());
    Case T_BPF:
        if (((Filter*)f) ->f2() < f1()) return f;
```

```
        else return new BPF(f ->f1(), f1());
    case T_Notch:
        cachedInput = f;
        return this;
    }
}
```

该参数是外加到这个滤波器的输入“值”（信号或滤波器），并且返回值既可能是一个信号也可能是滤波器-值对象，这依赖于它的输入。调用操作符将滤波器归入输入类：如果输入能够对电压（是T_Data类型）求值的话，那么看看是否该信号的频率是其以上的截止点，如果是，就返回该信号的值。此外，这个成员函数试图返回一个新的滤波器类型，其特性是其本身行为及其输入的组合。当用合适的输入变换函数描述时，就好像sine函数返回一个cosine函数一样。例如，任何低带通滤波器和高带通滤波器的组合将产生带通滤波器这样一个结果，如果高带通和低带通滤波器无公共频率通过，则该滤波器将会蜕化。如果在这些条件下返回单一基本滤波器类型较为困难，则滤波器就返回其本身，用另一滤波器配置作为输入。结果聚集对象将在模拟附近通过，就像它是单一的对象一样。如果该输入对Value对象是所有后向连接，则进一步引入evaluate，将返回Value对象以代替一个聚集（此回答是一个值，不是函数）。

注意，LPF::operator()(Value*)在它的参数之一的类型上含有一个switch语句，在本章开始时就给出一些忠告。开关语句的基本原理是防止公共的虚拟成员函数的扩散。如果使用虚拟函数方法重新执行operator()(Value*)函数，它有如下方式：

```
Value *
Filter::operator () (Value *f) {
    return f ->combineWithLPFoutPut(this);
}
```

（注意，此代码已经移入Filter类中，现在，它与所有滤波器类是相同的。）即，LPF将两个函数组合的任务传送到其他参与到组合中的滤波器。每个已存的case分支要求在某处由一个新的虚拟函数——“case T_LPF”引起：

```
LPF::combineWithLPFoutPut
BPF::combineWithLPFoutPut
HPF::combineWithLPFoutPut
Value::combineWithLPFoutPut
```

在这些函数中，整个代码与在case分支中的代码是相同的。所有else是等同的，switch的解比虚拟函数的解有较少的全局函数。这两种情形都添加了同样容易（或繁琐）的新的滤波器类型。

虚拟函数机制所选取的函数基于一个操作数的类。在更通用的说法中，似乎更为愿意的是选取的成员函数在操作中是基于两个对象参与的类型。这种功能（在C++中不能直接使用）称为多重方法，多重方法在后面的9.7节会做简明的介绍。

我们现在可以定义剩下的滤波器类型的类（BPF，带通滤波器的实现，见例5.18），并用它们构造和试验滤波器网络模拟。

例5.18：带通滤波器的实现。

```
Value*
BPF::evaluate(Value*input = 0) {
```

```
    Value *f = cachedInput ->evaluate (input);
    if (f ->f1() >f1() &&f ->f1() < f2()) return f;
    else return zero;
}
Value*
BPF::operator () (Value*f) {
    switch (f ->type ()) {
    case T_LPF:
        if (f ->f1() >f2()) return this;
        else return new BPF (f1(), f ->f1());
    case T_HPF:
        if(f ->f1() <f1 ()) return this;
        else return new BPF(f ->f1(), f2());
    case T_BPF:
        Frequency lowfreq = f ->f1();
        Frequency highfreq = ((Filter *)f) ->f2();
        if (f1() > lowfreq) lowfreq = f1();
        if (f2() < highfreq) highfreq = f2();
        return new BPF(lowfreq, highfreq);
    case T_Notch:
        cachedInput = f;
        return this;
    case T_Data:
        myType = T_Data;
        cachedInput = f;
        return evaluate ();
    }
}
```

例 5.19 表示了试验滤波器类的简单主程序。

例 5.19：试验应用滤波器的简单程序。

```
1  int main() {
2      Value *v = new Value(100, 1260);      // 在一个频率上的电压
3      BPF bpf(1000, 10000);                 // 一个带通滤波器
4      HPF hpf(1100);                        // 一个高通滤波器
5      LPF lpf(8000);                        // 一个低通滤波器
6      Filter *a;                            // 指向一个滤波器的指针
7      a = (Filter *)bpf(&hpf);              // 把一个带通滤波器加到
8      a ->print();                          // 高通滤波器上：结果？
9      (*a)(v) ->print();                    // 把电压和打印函数加到
10     a = (Filter*)(*a)(&lpf);              // 低通滤波器上：能得到双倍信息？
11     a ->print();
12     a = (Filter*)(*a)(v);                 // 现在，把电压加到所有输入，并且打
13     a ->print();                          // 印与低通和高通滤波器组合
14     lpf(&hpf) ->print();
15     return 0;
16 }
```

我们首先创建 3 个滤波器：一个是低通滤波器；一个是高通滤波器；还有一个是带通滤波器。低通

滤波器让低于 8 000Hz 频率的信号通过，高通滤波器让 1 100Hz 以上频率的信号通过，而带通滤波器让 1 000Hz 和 10 000Hz 之间频率的信号通过。我们把带通滤波器外加到高通滤波器上（第 7 行）并打印它（第 8 行），就会产生如下的结果：

```
BPF(1100, 10000)
```

即，把带通滤波器功能元件的应用加到高通滤波器功能元件之上，结果成了单一滤波器-值的对象，带通滤波器让 1 100～10 000Hz 之间的频率信号通过，其行为是用原始带通滤波器和用高通滤波器两者都是相同的。在第 9 行，这个滤波器的值外加一个信号并打印，产生如下结果：

```
Value 100 volts at frequency 1260
```

即，该信号频率（1 260Hz）落到带通滤波器（1 100～10 000Hz）的带通之中。在第 10 行，将此滤波器外加低通滤波器，并在第 11 行打印该结果，我们有

```
BPF(1100, 8000)
```

这个滤波器在第 12 行外加相同的电压信号，并且它仍然落在该滤波器范围之内：

```
Value 100 volts at frequency 1260
```

最后，我们组合低通和高通滤波器（第 14 行），并且打印带通滤波器的结果：

```
BPF(1100,8000)
```

这些应用程序设计的精髓是采用函数对象指针作为第一参数，接下来的参数是功能元件本身：

```
Value *apply (Value *object, Value *arg) {
    return (*object)(arg);}

Value  *apply(Value  *object, Value  *arg1, Value  *arg2)  {
    return (*object)(arg1, arg2);}
```

为方便起见，让我们也定义一个 printer 功能元件：

```
class Printer:public Filter {
public:
    Type type() {return T_Printer;}
    void print () { }
    Value *evaluate (Value*) {
        // 错误
    }
    Value *operator () (Value*v) {
        v ->print();
        return 0;
    }
} printerObject;
Printer *printer = &printerObject;
```

我们现在能够把低通滤波器外加到高通滤波器上，并对结果滤波器外加一个输入值，产生输出并放大了 3 倍：

```
LPF *lpf = new LPF(W1);
HPF *hpf = new HPF(W2);
AMPLIFIER *amplifier = new Amplifier;
```

```
Value *inputValue, *three = new Value(3);
apply(
    printer,
    apply(
        amplifier,
        apply(
            apply(lpf, hpf),
            inputValue
        ),
        three
    )
);
```

其语义与下面的语义是相同的：

```
amplifier((*((*lpf) (hpf))) (inputValue), three) ->print();
```

5.7 多重继承

有时，采用继承的单个线程是不够的，我们观察到新类可能需要继承两个或多个以上父类的特性。如果我们在编译时能够决定是哪一个父类的话，那么在单一导出类中就可以采用C++的多重继承来组合多个基类的行为。多重继承是一种强有力的特征，有丰富的语义，能够直接表示复杂的结构。当单继承（或者都未继承）丢失了重要的设计信息时，或者对问题的解进行折衷时，多重继承应当采用，但不要滥用。

动态多重继承在Flavors语言中是作为“混合插入件”而引入的，它在运行时对类型行为的“调谐”有用。用户可能组合所选的基类行为，为特殊的需求创建一个导出类，那么在应用中需要把创建的对象付诸于实现。在C++中，对所有可能派生的组合导出类在编译时必须加以说明，并且程序在运行时能够选取合适的类，同时创建该应用的对象。这种作法有可能导致导出类的组合信息爆炸。在Flavors中，这些类型能够建立得更加灵活：程序就能“混合插入”到被选取的行为之中，在运行时基类改变其基本行为是轻而易举的事。有关动态多重继承将在第10章加以介绍，这里只介绍静态多重继承。

一个窗口抽象的例子

在文献中一个通用的例子是使用多重继承支持终端的窗口/屏幕系统。这里，我们通过在终端上用文本编辑器显示文件内容的方式来讨论Window类：

```
class window {
public:
    window();
    virtual  ~window();
    virtual void addch(char) = 0;
    virtual vpid addstr(string) = 0;
    virtual void clear() = 0;
    virtual void scroll() = 0;
    ...
private:
```

```
    Color curColor;
};
```

该编辑器可在几个终端上交叉移植。这里有两种不同的 Window 导出类的种类，每种都采用了窗口技术。第一种称为 CursesWindow，是用 Curses[3] 包来实现 Window：

```
#include <curses.h>
class CursesWindow  :  public Window {
public:
    void addch(char);
    void addstr(string);
    void scroll();
    void clear();
    ...
private:
    WINDOW *cursesWindow;
};
```

第二种称为 XWindow，特别为 X 工作站窗口系统设计：

```
#include <X11/X.h>
class XWindow: public Window {
public:
    void addch(char);
    void addstr(const string);
    void clear();
    void scroll();
    ...
protected:
    Display *disp;
    Window window;
    XWindowAttributes  WindowAttributes;
    GC gc;
    Cursor theCursor;
    unsigned long WhitePixel, blackPixel;
    };
```

大多数 Window 成员函数是虚拟的，这意味着实际上它们是在导出类中实现的。其思想是类型对象 CursesWindow 和 XWindow 能够被实例化，然后通过编辑器传递，并且使用时就像它是类型 Window 一样。其他 C++程序能够用可移植 Window 接口来编写。

虽然其中一种的目的是为许多程序的使用提供一种通用 Window 的功能，但剪裁类 Window 需要编辑器也是合乎逻辑的。EditWindow 是具有特殊编辑器属性的 Window，诸如在屏幕上的第一行和最后一行：

```
class EditWindow : public Window {
public:
    int topLine ()      { return topLineVal; }
    int bottomLine()    { return bottomLineVal; }
    String fileName()   { return fileNameVal; }
```

3 Strang,John.Programming with Curses. Newton, Mass: Reilly and Associates, 1986.

```
    ...
    void scroll();      //除了执行窗口操作之外，还要处理
                        //topLineVal 和 bottomLineVal
private:
    int topLineVal, bottomLineVal;
    PathName fileNameVal;
    Editor *editorVal;
protected:
    Cursor theCursor; //特定的编辑游标
};
```

EditWindow 应该是 Window 的导出类，但也应该有像 CursesWindow 或 XWindow 一样的行为。然后，该问题就变成了如何建立类的导出关系。

这个问题的回答是，我们能够创建一个既能继承窗口技术特性又能继承编辑器体系结构的类，在单一的新的窗口类中的结果我们能同时使用：

```
class EditWnd_Curses:
    public EditWindow, public CursesWindow {
};
EditWnd_Curses aNewWindow;
```

这种简单的了解是足够了，即新的导出类继承两个父类的所有特性。但是，还存在两个问题：

（1）在新类上某些操作存在着二义性。例如，引用 aNewWindow.scroll()的含义是什么？是应当执行两个父类的 scroll 操作吗？执行的次序又是什么呢？

（2）该类的两个父类有一个公共的祖先类（Window）：它的实例数据在 EditWnd _Curses 实例中是应该被包含一次呢，还是两次？

关于第（1）点，与 C++中大多数面向对象程序设计语言不同，在 C++中没有如何处理这种二义性的缺省假定。语言却把此问题又扔给了程序员，允许使用语言已有的结构去处理它，这就避免了要添加语言新特征这样的复杂性来解决这个问题。面向对象程序设计语言嵌入到 Lisp 的一种概念，被称为“包装”（wrapper），并且在方法前后来处理这种二义性。C++解保存类似于 Lisp 语言的整个预先和事后条件结构的处理能力。关于第（2）点，C++有称之为虚拟基类的机制，竭力效仿 CLOS 类的单个实例语义，即在继承有向无循环图（DAG）中效仿多重外部特征的单个实例的语义。

注意，这些问题绝不是程序设计语言人工所为，而是在应用结构中名符其实地与应用相关的二义性，分辨这种名字冲突是设计上的问题，虽然 C++在编码中能够表达这种设计决策。

C++这样定义冲突操作符的说明是合法的。在含混的方式中只使用冲突操作符会产生致命的编译错误。一个好的忠告是在设计时应当了解这个问题，在编码后编译完成所有冲突的标识。下一节我们将详细讨论如何分辨这些冲突以及多重继承类的复制数据的问题。

成员函数的二义性

多重基类的成员函数的名字可能在导出类中发生冲突，用户可以添加代码来分辨这种冲突。关于成员函数 CursesWindow::clear 与 Window::clear 的冲突，因为后者并未做什么（它只是一个位置占位符），我们可以忽略它，并可进一步对 CursesWindow 进行操作。但是，类 EditWindow 和类 CursesWindow 的 scroll 成员函数也完成 EditWnd Curses 的 scroll 功能。如果两个函数必须调用，则调用的次序很重要。导出类 EditWnd_Curses 用自己提供的 scroll 操作，通过重写基类 scroll 函数来

分辨二义性。导出类 scroll 按我们要调用的次序调用成员函数：

```
class EditWnd_Curses:public EditWindow, public CursesWindow {
public:
    void clear() { CursesWindow::clear(); }
    void scroll() {
        EditWindow::scroll ();
        CursesWindow::scroll();
    }
    ...
};
EditWnd_Curses aNewWindow;
```

在 EditWnd_Curses 实例上调用 scroll 成员函数不再有二义性（在类域本身操作）。scroll 指定的实现是在对象的基类中按顺序调用 scroll 函数，在导出类上调用它的基类是不容置疑的——成员函数可能已经包含我们想要的任何代码，但在指定的次序中所有潜在的二义性操作符的执行，在面向对象程序设计语言中是公共的习惯用法。在 C++中，这种习惯用法依赖于非特定语言特征。

数据的二义性

一种分辨数据二义性的方法与分辨函数二义性的方法非常相同。考虑到：

```
class EditWindow:public virtual Window {
public:
    ...
protected:
    Cursor theCursor;
};
class XWindow: public virtual Window {
public:
    ...
protected:
     Cursor theCursor;
};
class EditWnd_X: public EditWindow, public XWindow {};
```

在上面的代码中，EditWnd_X 有两个称为 theCursor 的成员：EditWindow :: theCursor 和 XWindow :: theCursor 则

```
EditWnd_X::refresh() {
    ...
    // 错误；是 EditWindow::theCursor 还是 XWindow::theCursor?
    ... this ->theCursor...;
    ...
}
```

因为它是二义性的，所以是非法的。这种二义性能够用显式量化的方法来分辨：

```
this ->EditWindow::theCursor;      // EditWnd_X's EditWindow's cursor
this ->XWindow::theCursor;         // EditWnd_X's XWindow's cursor
```

虚拟基类

第二个问题是在另一个祖先类（多重导出类）中类的多重外观特征。我们通过使一些基类说明

以虚拟的方式来解决这个问题。这就意味着即使在导出的有向无循环图（DAG）中类 Window 出现好几次，它的数据在类 EditWnd_Curses 中也仅出现一次：

```
class window {
    ...
};
class EditWindow: public virtual Window {
    ...
};
class CursesWindow: public virtual Window {
    ...
};
class XWindow: public virtual Window {
    ...
};
class EditWnd _Curses :
    public EditWindow,
    public CursesWindow
{
    ...
};
```

缺省时，基类的数据在导出类对象中出现的次数与在祖先类中出现的次数一样多。如果一个类说明为虚拟派生的，那它就心甘情愿地与其他导出类共享基类，该基类应多次含有虚拟父类。如果基类的所有路径通过继承 DAG，采用传递虚拟派生的方式来启动该导出类，那么将存在该数据的一个拷贝。如果存有虚拟和非虚拟派生混合，则对每个非虚拟派生该数据将有一份拷贝，另加上对所有虚拟派生组合的一份拷贝。

注意，如果它们共享公共的虚拟的基类，并同时出现在同一个多重继承的 DAG 中，则这些类彼此违背了封装原则。不必惊讶，因为这些类通过使用虚拟派生的方式心甘情愿地共享它们的基类。任何一个基类变量的合并的发生都在程序员控制之下——既不是缺省行为也不是作为其他程序设计语言的强制行为。

避免虚拟基类中冗余基类函数的调用

现在，我们在 CursesWindow 或 XWindow 实例中就有了单一 Window 对象的信息，试问有关成员函数行为的相关问题。再来看看我们如何来分辨成员函数的二义性：

```
void EditWnd_Curses::scroll() {
    EditWindow::scroll();      CursesWindow::scroll();
}
```

我们在窗口上发现上述问题类似于刷新（refresh）和其他操作。可能要做的工作是按照合理的次序调用像命名函数那样的所有父类。EditWindow::scroll 和 CursesWindow::scroll 的依次向它们的父类 Window，在滚动（scroll）活动中请求它的参与：

```
void EditWindow::scroll () {
    ...  // 限于填入 EditWindow 方面素材
    Window::scroll();
}
```

```
void CursesWindow::scroll() {
    ...  // 限于填入 CursesWindow 方面素材
    Window::scroll();
}
```

注意，当 scroll 操作应用到类对象 EditWnd_Curses 且 Window::scroll 为两次调用时，这不可能是我们所要求的。对于 clear 函数，这有点不称职也不方便。对于 scroll 来说，实际上可能是一个错误。我们希望能够控制与导出类 CursesWindow 和 EditWindow 的执行代码有关的调用 Window::scroll 的顺序。

我们能够用程序设计约定来较好地调节这种顺序。EditWindow 特有的代码就产生了一个分离的成员函数_scroll：

旧的：

```
class EditWindow:
          pulic Window {
public:
    void scroll () {
      ...  //局部逻辑
      Window::scroll ();
    }
    ...
};
```

新的：

```
class EditWindow:
          public Window {
public:
    void scroll () {
      _scroll () ;
      Window::scroll ();
    }
protected:
    void_scroll () {
       ...  // 局部代码
    }
    ...
};
```

对 CursesWindow 的作法与之相同。现在，何时和按什么次序调节父类的活动，对于导出类来说可以有更多的选择：

```
class EditWnd_Curses:public EditWindow,
                     public CursesWindow {
public:
    void scroll () {
        // 这 4 个函数可按程序员所希望的次序运作
        EditWindow::_scroll ();
        CursesWindow::_scroll ();
        _scroll ();    // 或许是可选的
        Window::_scroll ();
    }
protected:
    void_scroll () {
        ...     // 任何添加的工作导出类必须做
    }
    ...
};
```

这里，成员函数 EditWnd_Curses :: scroll 指示基类 EditWindow。首先应当运行滚动（scroll）函数，接着运行 CursesWindow。然后 EditWnd_Curses 调用它自己的滚动逻辑函数（如果有），这完全类似于对 EditWnd_Curses 的分割。最后，把所有事情系在一起调用基类 Window::_scroll。其他次序由程

序员来指定。

引入辅助的_scroll 函数，我们称之为“多重继承函数意义不明确的习惯用法”。

虚拟函数

虚拟函数像单一继承一样，在多重继承中是有用的。其行为是对在单一继承所发生的行为的直观扩充：函数调用由该对象本身类型所决定，不必由指向该对象的指针类型来决定。

考虑上面的 Window 类的例子。变量的说明是 Window*，可以指向层次结构中任一类的一个实例。而指针可以用于调用一个成员函数，如果该函数是虚拟的，那么函数调用将由指向窗口对象的类型来决定。假定成员函数 scroll 在基类中说明为 virtual：

```
class Window {
pubic:
    ...
     virtual void scroll () = 0;
    ...
};
```

调用虚拟 scroll 操作，在运行时被迅速调遣到正确的对象上：

```
int main () {
    EditWnd_X anXEditWindow;
    EditWnd_Curses aCursesEditWindow;
    Window *windowPointer = &anXEditWindow;
    windowPointer ->scroll ();      //调用   EditWnd_X::scroll
    WindowPointer = &aCursesEditWindow;
    windowPointer ->scroll ();      //调用   EditWnd_Curses::scroll
    return 0;
}
```

下面我们将介绍如何执行虚拟函数的工作，它属于单继承的情况。

清除指向多重继承对象的指针

考虑以上述类 EditWindow 和 XWindow 为样板的情况，而 EditWnd-X 为

```
EditWnd_X *pe = new EditWnd_X;
XWindow *px;
...
px = (XWindow*)pe;          // px 现在包含了与 pe 不同
                            // 的物理地址
px = pe;                    // 与前一个一样的结果
pe = 0;                     // pe 设为整数零
//pe = px;                  // 错误，需要抛弃
pe = (EditWnd_X *)px;       // pe 重存入原始值
```

这里，单个 EditWnd-X 对象通过建立 pe 指向该对象的公式来初始化。我们消去 pe 进入指向 XWindow 的指针，并把其值拷贝到 px。因为 C++允许类对象指向任何公共导出类的地址（此导出类说明为基类型），所以消去是合法的，并在 C++语言层上有直观的语义。但是，如果发生在“底层”，此工作就不直观了，并且要承担检测类型的工作。

在多重继承中，在内存不同的地方启动对象不同“部分”的表示。如果 px 指定为 pe 的值，px

必须指向原始的 EditWnd_X 对象的“XWindow 部分”，所以成员们能够被合理地间接引用。这里会产生一些副作用，我们考虑下面的代码：

```
void an_x_function (void *p1, void *p2) {
    if (p1 == p2) printf("p1 == p2 \n");
    else printf ("p1!= p2 \n");
}
EditWnd_X*pe = new EditWnd_X;
XWindow*px = 0;
px = pe;
an_X_function (pe, px);
```

在 an_X_function 中平等地测试在大多数 C++实现中会造成失败。消除一个类指向另一个类的指针，然后对第三个类型(这里是 void*)不保证产生类似于直接从类指针指向第三个类型的相同的结果。这有点像是在消除 float 的模拟，然后对 double，其中从 float 到 double 直接产生不同的消去值。

对于空指针要作特殊的处理。在 C++程序中，消去不同类型的指针以确保空指针仍能成功地继续进行比较：

```
EditWnd_X *pe = 0;
XWindow *px = 0;
if (px == 0)  ...      // 测试成功
pX = pe;
if (px == 0)  ...      // 测试成功
```

5.8 继承的规范格式

需求的改变和技术的更新都会导致程序结构的变化。合理地调节继承结构将有利于长期的维护。类继承的约定可以使程序容易理解，并确保对象的操作进行多态的处理。

例 5.20：继承的规范格式。

```
class Base {
public:
    // 基类遵循传统规范格式
    Base();
    Base& operator = (const Base&);
    virtual ~Base();
    Base(const Base&);
private:
    ...
};
class Derived:[virtual] [public|private] Base {
public:
    Derived ();
    Derived& operator = (const Derived&);
    ~Derived();
    Derived(const Derived&);
    void foobar() {
        ...
```

```
        _foobar();
        ...
        Base :: foobar();
        ...
    }
protected:
    void_foobar () { ... }
};
```

例5.20提供了C++类派生的规范格式。一种方式是应该考虑采用缺省方式使所有派生虚拟，这些合理性的基础与多重继承的语义有关，要了解有多少个Base实例数据应当保留在与多个独立路径Base家族有关的对象中。我们来考虑基类Window，其中有两种类型的导出类：一种是依赖于窗口系统的导出类；另一种是依赖于应用的导出类。一方面，我们要开发与X窗口系统不同种类的微软窗口系统；另一方面，从应用的观点来看，这两种系统的行为特性是类似的，但实现却是不同的。同样地，我们要开发为字处理器使用的特殊种类的窗口，另一个是为编辑器使用的窗口，这依赖于我们如何引入用户的接口。我们可以使用类似于如下方式的继承DAG：

```
class window {
    ...
private:
    short windowID;
};
class XWindow: public Window {
    ...
};
class MSWindow: public Window {
    ...
};
class WPWindow: public Window {
    ...
};
class EditWindow: public Window {
    ...
};
```

现在如果我们要

```
class X_WP_Window:public Xwindow, public WPWindow {
};
```

则会存在有多少个Window::WindowID域？答案是两个。WPWindowManager可能保持与字处理窗口的联系，而分开的XWindowManager保持与X系统窗口的联系，每个可能都有它自己的窗口对象所应具有的ID域。这就意味着每个对象应当有每个窗口系统的ID域——在这种情况下，每种类型自己应有的窗口管理者。如果类Window在链接列表上有自身的维护结构，则有些事情的作法就都是相类似的了：

```
class Window {
public:
    ...
    void insert (Window*);
```

```
    ...
private:
    Window *previous, *next;
    ...
};
```

如果每个窗口系统要求有“自己的”窗口的链接-列表-编辑器，则虚拟派生就不必使用。不幸的是，当我们做派生第一层（单层）而不是派生多层时，这个决定就形成了。只要仔细考察，有一定的预见能力和洞察能力（有时是事后认识的）就能够帮助我们在程序演变避免重构继承结构。我们总不能老是猜测，或者摸着石头过河，这些问题事先仔细地考察是完全值得的。

下面介绍的银行账户应用类是使用虚拟派生的较好的例子（在本章结束处有简短的应用说明）。我们有一个基类 Account、导出类 CheckingAccount 和 SavingsAccount：

```
class  Account {
public:
    virtual void  print_statement () = 0;
    ...
protected:
    // 公共低层文件访问由导出类使用过的函数
    void file_read(const char*);
    void file_write(const char*);
private:
    AccountNumber accountNumber;
};
class CheckingAccount:public virtual Account {
...
}
class SavingsAccount:public virtual Account {
    ...
};
```

考虑以下的派生：

```
class CheckingSavingsAccount:public CheckingAccount,
                                public SavingsAccount {
    ...
};
```

下面是我们使用虚拟派生的两个原因：

（1）每个账户对象有它自己的标识（账号）；我们不要 CheckingSavingsAccount 有两个分开的账号；

（2）该派生是作为组合行为描述的，而不是组合描述。在类 DAG 的顶端上，在初始派生的层次上再一次需要预见能力。多重继承需要推迟这些决定，这可能导致设计工作的重复，并将大范围地触发重新编译的代码。

上述规范格式的例子也需要从导出类成员函数到同名的基类成员函数的分开调用作出特别的努力。此“规则”针对这样的函数：此函数的一部分是发生在 protected 节，用导出类作局部处理，而与它们的基类的相互作用在导出类的“真实”成员函数的主体中可以被调节。这些步骤的采用也支持多重继承，更为详细的说明在 5.7 节已作过讨论。仅当导出类函数正在调用同名的基类函数时，这样的步骤才是必须的，并且将来有可能虚拟派生。

当使用这种习惯用法时：这种习惯用法的使用应当用两种考虑来调和。首先，如果确切知道绝不使用多重继承，则从基类中派生虚拟是不必要的。第二，虚拟派生可能会影响性能，需要在普遍性和微妙的保存之间加权折衷。

下面是一些创建继承结构的其他准则：

（1）当使用公有继承时，大多数基类成员函数应当说明为 virtual。导出类的这个说明是基类的特化。它确保了基类的定制将重写基类的缺省行为，甚至当导出类对象出现在期望基类对象的上下文中时。例如，如果 SunWindow 是从 Window 派生出来，那么我们就应能提供一个期望在 Window 处的 SunWindow。为了使编译实际调用 SunWindow 成员函数，该函数必须说明为 virtual，所以编译能在运行时安排选择正确的函数运作。

（2）当使用私有继承时，大多数基类成员函数不应当说明为 virtual（将在 7.5 节进一步加以讨论）。

（3）如果打算消除基类指向导出类指针的指针，则虚拟继承不能使用。若是这样，那就很少应当做些什么；通常的建议是把一个非虚拟函数看做是虚拟的，或者那个虚拟函数需要在层次结构中添加这个类。

（4）如果不特别关注性能，或者如果该类有机会在多重继承中参与开发，那就要使用虚拟派生，这取决于下面的考虑：① 看一看在基类中所描述的信息。如果要以每个实例为基础进行复制，特别是在多重继承下进行，那么就不要使用虚拟派生。② 如果该类数据的一部分适合这条规则，而并未去做这部分工作，那么要重新考虑设计。要做出什么样的正确表示，才是多重继承？③ 是否愿意允许基类的封装被违背？虚拟派生允许在微妙的方式中违背封装性。

5.9 例子

1. 队列迭代器（queue iterator）例子

```
class Queue {
friend class QueueIterator;
public:
    void *leave() {
        void *retval;
        if (head == tail) retval = 0;
        else {
            if ( --head < 0)head = max - 1;
            retval = rep [head];
        }
        return retval;
    }
    void *first() {return rep[head]; }
    void enter(void *V) {
        if ((head +1) & max == tail) {/ * error -- full * /}
        if (--tail<0)tail = max-1;
        rep[tail] = V;
    }
    int empty() {return head == tail; }
```

```
    Queue() {rep = new void*[max]; head = tail = 0; }
private:
    enum Constants { max = 10 };
    void **rep;
    int head, tail;
};
class QueueIterator {
public:
    QueueIterator (const Queue& q): queue(q), current (q.head) { }
    int next (void *&i) {
        if (current == queue.tail) {
            return 0;
        } else {
        if (--current < 0) current = Queue::max - 1;
        i = queue.rep[current];
        return 1;
        }
    }
    void reset () {current = queue.head; }
private:
    const Queue &queue;
    int current;
};

int main() {
    int one = 1, two = 2, three = 3, four = 4, five = 5, six = 6,
        seven = 7, eight = 8, nine = 9, ten = 10, eleven = 11, twelve = 12;
    void *i;
    Queue a;
    a.enter (&one);
    a.enter (&two);
    a.enter (&three);
    QueueIterator qi = a;
    while (qi.next(i)) printf("&d", (int) (*((int*)i)));
    printf("\n");
    a.enter(&four);
    a.enter(&five);
    a.enter(&six);
    printf("leaving:"); i = a.leave();
    printf("&d", (int) (*((int *)i)));
    i = a.leave();
    printf("&d", (int) (*((int *)i)));
    i = a.leave();
    printf("&d", (in t) (*((int *)i)));
    i = a.leave();
    printf("&d", (int) (*((int *)i)));
    i = a.leave();
    printf("&d\n", (int) (*((int *)i)));
    a.enter(&seven);
    a. enter(&eight);
```

```
    a. enter(&nine);
    a. enter(&ten);
    a. enter(&eleven);
    a. enter(&twelve);
    QueueIterator qj = a;
    while (qj.next(i)) printf("&d", (int)(*((int*)i)));
    printf ("\nleaving:");
    i = a.leave();
    printf("&d", (int) (*((int*)i)));
    i = a.leave();
    printf("&d \n", (int) (*((int*) i)));
    qj.reset();
    while (qj.next(i)) printf("&d", (int)(*((int*)i)));
    printf ("\nleaving:");
    i = a.leave();
    printf("&d", (int) (*((int*)i)));
    i = a.leave();
    printf ("&d\n",(int) (*((int*)i)));
    return 0;
}
```

2. 简单的银行账户（banking account）应用类

```
#include <fstream.h>
class Account {
public:
    Account ();
    Account (double penalty, double minimum,
                        const char *name = "Account");
    ~Account ();
     virtual void deposit (double) = 0;
     virtual void withdraw(double) = 0;
     virtual void print_statement () = 0;
protected:
     void file_write (const char*);
     void file_read (char*);
protected:
     void _deposit(double);
     void _withdraw(double);
     void _print_statement();
private:
    double penalty;
    double minimum;
    double balance;
    char*name;
    ifstream fpr;
    ofstream fpw;
};
class InterestAccount : virtual  public  Account {
protected:
    InterestAccount (double penalty,
```

```
                    double interest, double minimum,
                    char*s = "InterestAccount");
   ~InterestAccount();
public:
   void deposit (double);
   void withdraw(double);
   void print_statement();
   double calculate_interest();
protected:
   void _deposit(double);
   void _withdraw(double);
   void _print_statement ();
private:
   double interest;
};

class SavingsAccount :public InterestAccount {
public:
   SavingsAccount (double penalty,
                   double interest,
                   double minimum,
                   char*s = "SavingsAccount");
   ~SavingsAccount();
private:
};
class CheckingAccount :virtual public Account {
public:
   CheckingAccount(double penalty,
                   double minimum,
                   const char*s =  "CheckingAccount");
   void deposit(double);
   void withdraw(double);
   void print_statement();
protected:
   void _deposit(double);
   void _withdraw(double);
   void _print_statement();
private:
};

class InterestChecking Account : public CheckingAccount,
                                 public InterestAccount {
public:
   InterestCheckingAccount(double penalty,
      double interest,
      double minimum,
      const char*s = "InterestCheckingAccount");
   void  deposit(double);
   void  withdraw(double);
   void  print_statement();
```

```
protected:
    void _deposit(double);
    void _withdraw(double);
    void _print_statement();
private:
};
```

练　习

5.1 讨论当使用多态性作为抽象机制时的继承。

5.2 虚拟函数在与抽象数据类型的类体系构造模型上可以认为是支持面向对象程序设计的基石。多态函数在过程体系构造模型上是支持面向对象程序设计的基础。一个熟知的引用例证是多态 sort 函数，它能够对任何种类对象的向量进行排序。在 C++中，你能做多态函数吗？将 C++方法与其他面向对象程序设计语言的方法加以对照。

5.3 在 C++中，模板（见第 7 章）是创建多态函数的一种方式：它们是提供编译支持合理类型或类型集合上工作的函数。你在什么地方使用模板以替代继承和虚拟函数？使用模板比使用虚拟函数更具有优势的地方是什么？模板比虚拟函数有更多的限制吗？为什么有，为什么没有？

5.4 在以下程序中使用诊断程序（debugger）跟踪函数调用，其中 Atom 结构是本章描述过的。

```
Atom  *lexAtom(String & s) {
    GeneralAtomg(s);
    return g.transform ();
}
void parse (String & S) {
    for (Atom *a = lexAtom(s); a; a = lexAtom(s)) {
        ...
    }
}
```

5.5 重新实现 5.5 节中 LAPD 类的消息处理，要求不要使用缓冲区拷贝（传递指向类构造函数的指针，相对于不同种类的消息和不同层的协议）。

5.6 实现 5.6 节的 Amplifier 类。

5.7 在 C++中，用功能元件习惯用法设计并实现类的集合，执行符号代数。试图添加微分器和积分器，在无约束变量的公式上作符号计算。当所有边界条件和参数都合适提供时，该类应当使用牛顿（Newton）方法、龙格-库塔（Runge-Kutta）综合方法等。

第6章 面向对象设计

前面几章已经讨论了C++继承和虚拟函数的语法和语义问题，这些都是面向对象程序设计的基础。这里，我们回过头来再由设计的观点和更远一点的体系结构观点来看待这些理念。系统发展的关键在于程序的可理解性，理解程序的基础是要有一个好的设计。对C++来说，好的设计的关键是理解类、对象和成员函数要做些什么，不做些什么。

对象范型很适合构造各种各样复杂的系统，大多数复杂的软件应用都能用类和对象的抽象方法获取以下这些特征：资源、事件和应用领域的“有形资产”。其他技术，如功能分解和数据流分析，在某些领域中有很好的业绩，但却不可能广泛地应用。面向对象分析对大多数系统来说是一个很好的起跳点，其他技术可以作为面向对象实现策略的补充。

面向对象系统构造技术超越了管理结构的技术领域，提供了与用户沟通的工具，从而能获得预期的最终产品。

对象范型与一般范型一样，包含了体系结构规则和指南。这些规则限定了设计和程序设计活动，但这些限定是为了保持设计和长期维护的一致性。这些限定首先为创建和管理抽象服务。对任何采用的设计方法，所建立的抽象应尽可能地处理事务的复杂性。对象范型的抽象技术集中在专有名词（术语）、资源和应用领域的抽象方面。其他许多设计方法则集中在产品的解和实现上。所幸的是，对象范型提供了实现技术——面向对象程序设计，它能直接获取实现中的抽象结构，把从应用分析到设计以及实现紧紧联系在一起，以帮助实现结构与应用的基础、稳定性和无损性方面的抽象相适应。这种在问题的特性稳定方面和解的结构之间的对应是系统支持对象范型的基础。我们称其为**同构结构原则**，并且它也是在封装改变的前提下对象和类的能力的核心。

本章探讨的软件从系统地阐述设计抽象观点出发，展示C++语言如何可在源代码中直接地获取这些抽象。同时，也介绍一些C++程序设计“不成文的规则”，对面向对象C++程序的灵活性、高效率和优美性提供技术和指南。本章要添加类、对象、类型和实体以及它们之间相互关系的一些精确定义。通过对这些基础性知识的介绍，本章就能把注意力转移到设计过程本身上来。接下来，本章将讨论建立类抽象和它们之间结构关系的设计规则。如果本章建立了面向对象设计的基础，那么我们就能过渡到更高级的习惯用法上。

6.1 类型和类

类通常用创建新的类型并把这些新类型引入到程序设计语言的方法来刻画。而大多数（虽然不是全部）语言把“类型”的概念看成是附加在语言中变量特性的直观的集合。例如，我们知道，整

数可以附加在加法、减法、乘法和除法上；在有串变量的语言中，我们知道能够被耦合，可以搜索子串等等。

这里，让我们从某种历史观点来作一番探讨。最初，类型概念是作为集合理论特性的形式化基础而出现的。这样，type integer 的含义就是所有整数的集合；它覆盖了在集合成员上的有效的操作和特性，诸如应用集合的闭包。如果一个集合包含在另一个集合之中，则这个集合就称为“子集合”。每一个集合都用一种类型描述。较小集合的类型描述被称为较大集合的子类型。例如，整数是 Real 的子类型，因为实数集合包含了整数集合。

在开始研讨程序设计语言时，起初试图在语法中模型化数学表达式。这就是传说中最早的程序设计语言 FORTRAN。FORTRAN 函数是模型化的数学函数；它的类型就是数学类型，其目的是在程序设计语言中直接模型化问题领域的抽象模式。早期的大多数数值计算集中在工程和数学领域，那时它们的实体和类型都反映在语言的变量和类型上。但是，程序设计语言必须对表示和行为作一些让步。例如，数字集合不可能真正地无限大，但在实践中用机器字的长度可以加以限制。

最终，“类型”变成了方便的记号，失去了理论产物的大部分含义。例如，许多语言（包括 C，C++以及经由 FORTRAN90 通过的 FORTRAN 版本）允许在字符上进行算术运算。在应用语义中，虽然有模糊形式化逻辑存在，但在真实语言中仍是极其有用的。在大多数程序中的大多数函数并未精确地实现数学函数，所以形式化的根已经失去了。

类的概念也不是很新的，至少可以回溯到上个世纪初的 Principia Mathematica[1] 早已提到过了。那时，把类描述成“仅仅是作为符号或语言学家的方便”。经过长期的努力，获得了许多类术语的真正含义，并把这些含义保留至今，同时也覆盖了今天我们所说的“类型”的含义。

在 C++中，可以创建自己的类，并在传统的程序设计语言中用类领悟到类型的意思，也可以在任何两个类之间定义转换，使之含有应用建模的含义。当定义从 double 转换到用户定义的 Complex 的意思时，它不总具有转换的含义。比如说，有 List 到 Set 的含义，或者有苹果到橙的含义。在设计期间，这些问题应当加以处理，而程序设计语言仅提供有限的帮助。这样的问题最好从设计观点出发，用促进合适语义的设计规则和更好的设计美学方法来解决。

所以我们有了在应用领域中的传统的类型和可以表达相应解的类。对好的面向对象设计来说，我们要把应用领域的实体映像到对象，而把类型映像到解领域的类上，正如图 6.1 所示的那样。应用领域实体是在应用中所关注的概念：在银行系统中的交易、纸币、账户和出纳员；在电话通信系统中的电话、呼叫、线路和开关。我们把这些单个的实体概括成类型，即，通常我们可以谈论交易、账户、电话和呼叫，而无须在工商银行谈论张三的账户或者谈论具体的李四的家里电话。这些应用领域概念映像到解领域的软件抽象：类型变成了类，而实体变成了对象。

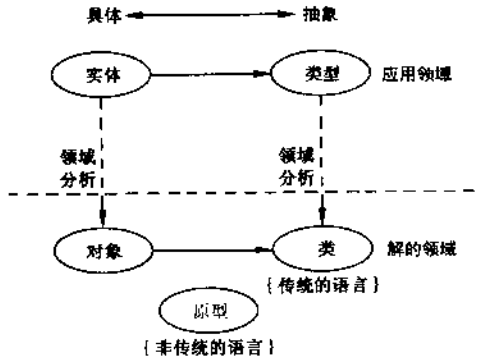


图 6.1 领域集合关系

只要是对 C++大部分合适的这种模型，构建就不

1 Whitehead，A.N.，and Russell，B. A. W. Principia Mathematica. Cambridge，Mass.：Cambridge University Press, 1910.

仅仅是可能，而成为现实的问题。如果类本身是一个对象，像 Self 或 Smalltalk 那样，实体和类型的概念能够作为对象来实现。同样地，程序设计语言构造能够用于从应用领域中同时获取实体和类型信息：当用于表示类型时，这样的对象就被称为原型或样本。本书选用后一个术语，而“原型”这个字已变成了重载的术语（见 6.2 节）。样本用于创建新的对象模型，既可以拷贝又可以执行原始成员函数，并返回一新的实例。在 C++的面向对象程序设计中，我们将在第 8 章介绍用 C++习惯用法建模一个样本方法。

除了如图 6.1 所表示的关系之外，应用领域类型之间的其他关系必须到解领域中获取。如果一个类型是另一个类型的特化——DigitalPressureMeter 和 AnalogPressureMeter 是更抽象的 PressureMeter 的每个特化——我们可在解领域中用继承来获取。从 PressureMeter 衍生出来的公共类 DigitalPressureMeter 导致了展示 PressureMeter 的特性，在两种抽象之间的应用领域获取的关系。我们能够把 Spring 对象放到 AnalogPressureMeter 内，就好像我们在真实世界中找到的一样。在绝大部分时间里这样的转换就是这样直接。因为问题领域和解领域之间是**同构的**，所以使用面向对象设计是非常合适的。这两个世界（领域）之间的完备的映像，可以用封装的变迁来支持得体的演变。我们在类的结构中反映出应用的结构。真实世界的整个结构通常比它的部分行为细节（反映在成员函数变化中）的变化要少，或者比它的部分如何实现（反映在它的内部数据结构和类的算法上）的变化要少。

公有继承是用于获取 C++中应用领域专门化（特化）关系的语言构造。公有继承、信息隐藏和虚拟函数是面向对象程序设计的支柱。用从 PressureMeter 衍生的公共类 AnalogPressureMeter 方法，我们认定类 AnalogPressureMeter 将在它的公共接口上展示所有的 PressureMeter 行为。其代码如下：

```
class AnalogPressureMeter: public PressureMeter {
public:
    ...
    // 从 AnalogPressureMeter 对象的用户观点来看，无论什么
    // 在 PressureMeter 中的公共动作几乎都在这里说明了
    ...
private:
    ...
};
```

但是，在 C++中衍生的缺省模式是私有派生，即基类成员函数仅能在导出类中说明，而不能由导出类外部用户来说明。私有继承比可重用实现技术的设计支持技术要少，这些将在第 7.5 节中单独进行讨论。公有和私有继承之间的不同之处我们在第 4 章已经讨论过了，这里不再重复。

在真实世界中，对象关系比单独用特化和包含的表达更复杂，并且已经出现了描述这些关系的术语词汇。借助于认知科学和实体-关系信息模型（ER 模型），我们用 IS-A 术语表示特化：DigitalPressureMeter IS-A PressureMeter。术语 HAS-A 用于描述包含：AnalogPressureMeter HAS-A Spring。但其他关系是怎样的？AnalogPressureMeter 是如何 LIKE-A AnalogVoltMeter？Manager，Secretary，DeptHead（部分领导），VicePresident，AdminAsst 和 CopyMachine 之间是什么关系？根据面向对象设计的精髓，我们要在实现结构中获得这些应用关系。在面向对象程序设计语言中，正当地获取这些关系需要一些附加的洞察力，需要用语言的习惯用法支持创建一个具有良好结构的程序。本章将提供这样的习惯用法。

面向对象程序设计语言在问题领域创办了新的 IS-A 关系。继承是根据它们的 IS-A 关系帮助组

织抽象的技术，而公共特性的重用很容易覆盖类似的类。但可重用的机遇与 IS-A 无关，并且仅与应用的特性弱耦合。C++也表达了代码解领域的优化和代码的可重用性。本章将帮助设计者区分这两种类型之间的关系。与解领域有关和与应用领域有关之间的结构权衡依赖于系统的维护能力和体系结构的完美性，是在工程上需要考虑的关键问题。

忽略维护应用领域形式化的表述和解构造之间的紧密关系，将会导致不可变更的实现的现象的发生，理解上也很困难。当类与子类型（或 IS-A）彼此相关时，这个问题变得非常有意义，这也是本章要讨论的。在讨论这些理念之前，让我们为基本的面向对象设计打下一些基础。

6.2 面向对象设计的实践活动

无论使用什么技术，对应用的深层次的理解是任何设计成功的关键。用面向对象技术来设计也不例外，但面向对象技术可使工作更容易些，因为在应用中（如电话和账户），系统构造块是从用户看得见的东西中派生出来的，而不是从解的领域（数据库、分层传达和磁盘驱动）中从实现者使用的事物中导出。实现上仍可以使用算法和工具，但这样的工具总是在用户理解事物的上下文中进行操作。

面向对象系统（或任何系统）的设计者关键要与最终用户有良好的工作关系，合理地假定有关系统的目的和操作。这里，原型和迭代开发能够帮助你。在提交最后的决策之前，通过几个可能的试探设计过程可以剔除无用对象，以帮助开发者进行迭代。一旦主要的抽象被用户所理解（满足用户需求），开发者就能开始使用开发工具进一步实现。

这些实践活动导致开发过程与自顶向下和结构设计技术有着本质的不同。这种区别是很重要的，适应用户的需求，学习迭代开发是一种思维方式的改变，也是面向对象的设计能否成功的秘诀所在。

要保持头脑中这种思维方式，这里我们列出下面一些面向对象设计的实践活动：

- 在应用领域中标识实体　实体可能是资源、事件和其他好定义的专门知识或特长领域。实体通常以问题叙述的名词提供给应用的最终用户。例如，如果开发飞行控制软件包的有见识的用户希望在飞机上添加（方向）舵，那么在飞行控制软件设计中就应该加上 Rudder 实体。

  在这个考虑的层次上，实体是一般的“类”实体还是具体的实例要求并不严格。作为实体纳入的这两种层次抽象之间的进一步区分被转换到实现结构中。
- 标识实体的行为　每个实体提供一些应用上的服务，既可以用任务的集合来执行，又可以通过其他实体来控制。这些服务是实体对应用所负有的职责。所有需要的服务和职责必须被标识，并且要与适当的实体相关联。
- 标识实体之间的关系，特别是子类型（IS-A）关系　如果一个实体是更一般实体的特化情况，那么这种关系应当表示，并且这种表示有助于在实现阶段被使用。在这种实践活动中的有用的辅助工具是事务图（transaction diagram），这部分内容将在第 11 章中加以描述。其他重要关系，诸如一个实体包含在另一个实体内的关系，也将在本章后面加以讨论。
- 由实体建立一个 C++设计结构　初始定义的实体是 C++类和对象的较好的候选者。具体的实体将映像到运行程序中的对象上，实体（类型）的概括变成了类。这样类的名字应该反映在应用领域副本的语义上。

在实现中子类型关系大都反映继承的层次结构，采用虚拟成员函数描述。有时，在实现中实体子类型关系用转发（forwarding）替代继承更合适。转发在本章后面会加以描述。

某些应用领域实体在实现中可以映像到类和对象，而对其他范型，则映像到抽象。例如，传统的数据库技术在分析应用期间，可以适当地建模在一些数据抽象的发现上。

- 实现　将类数据结构的细节完成，新的私有函数添加到类中去，并且编写好成员函数。在实现阶段可能要对类和成员函数单个地进行测试。
- 进行调整　因为某些部分的函数调用和自变量拷贝的费用太昂贵，所以为了提高效率，一些函数可能要说明为内联（inline）。你可能会发现在某些实现部分之间的共同之处，而在分析阶段本身却不明白。这里，我们可以用私有派生使得代码可以被重用。
- 测试　对系统进行黑盒测试。

注意，这些实践活动的步骤并不是按顺序的。这些活动按选择（尝试）法次序执行。有时，在实践中甚至是迭代进行的。实现成功的关键在于迭代和抛弃错误，从一开始就需要不断地改进。

在面向对象的开发中，传统的开发步骤之间的边界变得过分注意细节。早期实现（原型）能对设计阶段增加深度和洞察力，就好像一些设计决策可以规定一些实现一样。

注意，技术的关注点是在设计产品本身，它比每个程序流（对象）之间的交互作用更重要。当然，对象之间的相互作用也是很重要的，它可以告诉我们要做些什么事。一个系统结构的完整性来自于它的接口结构；对象接口建立了管理封装变迁及合理演变的结构。

通过分割系统而产生的系统面向对象分析与从结构中产生传统的函数分解技术有很大的不同，后者集中在系统片的交互作用上。在函数分解技术中，函数是遵从高层设计决策的人工产物。在高层分量之间，函数被建立在交互分析控制之外。通常增加的低层设计决策不明显或者与复杂的应用紧密相关。如果我们的关注点首先是在软件抽象上，其次是在相互作用上，那么设计将会合理地演变下去，也更容易理解。

考虑面向对象设计对公司的模拟：你的老板选择你和其他几个人共同在一个小组中工作，解决一个工程问题。老板就是系统设计者，选择你和你的经验，而你提供你在团队中的能力，并为老板服务。老板为这个小组制定一个目标，并且让你去达到它；不存在精确地进行交互的规范说明。你是所有彼此交互的自主对象，完成整个系统任务。

6.3 面向对象分析和领域分析

面向对象分析活动是分析问题的技术，根据某一方面问题的经验把该问题划分到抽象和封装中去。找出超越现有应用的范围，拓宽抽象对适合整个应用领域是特别重要的。给设计活动添加这种观点以刻画新的共同活动集合的特征，就称为**领域分析**。

为什么要拓宽设计范围

让我们来看一看对东方 123 号飞机签订的自动驾驶软件设计的一个合同。我们可以发现一些合理的抽象——方向舵、副翼、发动机等等——实际上是把它们集成到一起控制飞机的起飞、飞行和降落。

代之以东方 123 号的设计，我们需要拓宽抽象的力度，就好像我们不是对东方 123 号飞机进行

设计，而是进行假想的、抽象的东方123号飞机设计。这种假想的“抽象飞机”表示了我们所感兴趣的领域特性。这样做有3个理由：

（1）拓宽设计范围，准备处理新的抽象，类似于对这些现有的应用进行处理。公司签订东方123号飞行控制软件就很可能与将来签订飞机合同类似。我们要重用东方123号软件，以满足新的合同要求。

（2）拓宽我们的设计范围可以帮助系统开发。如果东方123号采用了新的发动机，新的液压系统和新的罗盘，那么在遇到需求改变时，我们就要尽可能少地改变已设计好的软件。如果该设计抽象把有关当前的发动机、液压系统、罗盘等等特定经验减到最少，那么当新的部件引入时就可使影响降到最低。帮助系统开发演变一种技巧是根据其他抽象互相作用的方法来刻画抽象特性，以替代如何进行内部实现。如果罗盘通过3个电信信号与飞行控制硬件相互作用的话，那么我们应该根据3个相应的特性刻画罗盘特性。新的罗盘将很有可能具有3个触头，并且我们的软件也很可能在需求改变时不受影响。

（3）拓宽我们的设计范围可以帮助系统最好地履行希望满足的需求。拓宽系统抽象的范围，我们能使系统“覆盖更多的需求基础”。如果最终系统未达到某些顾客期望的标准，那么拓宽设计比单一观点的设计优化更容易调节所期望行为的偏差。

如何拓宽领域

如何做我们才能拓宽迅速恢复变迁的抽象呢？一种方式是采用“长胖”的类接口，以提供所有合理的服务。例如，系统结构假定飞机有一台发动机，还是有几台发动机？我们可以采用核心动力部分建立的抽象是发动机向量的方法，给初始设计增加一些“胖”的类接口；东方123号采用单一元素向量是一种蜕化设计。这里，总开销和大部分向量对东方123号飞机来说是超负荷的，但多台发动机的设计可以使设计简化。基于概括原因而增加的类行为被称为**胖接口**。

研制胖接口是一种长期的概括技术。通常，过程线性代数库包括10个参数的子程序，其中的大部分是不用的。这样的抽象对学习和理解都很困难。但它是少数使人满意的技术之一，能够使用过程抽象，大都无须折衷协调。

用面向对象设计，它甚至与较好的作法正好相反：我们可从更一般的抽象去掉一些行为。东方123号的发动机分析可以显露许多能够被操纵的层，但在发动机抽象的接口中代替了每个可见层，它最好既能把几个层压缩到更抽象的概念中（浓缩），又能把单个层的外观隐藏起来。如果发动机有两个汽化器（这是控制燃料流和空气流进入发动机的装置），那么这两个汽化器可以组合成单一的抽象汽化器装置。用以作为设计工具的事务能够帮助显示这样的通用性；这些理念将在第11章中加以讨论。如果东方123号有极好的手动点火减压活门的话，则发动机的通用特性并不会显示出来。所以替代胖类，我们要用通用的“瘦类”，因为对于实体和类型，它并不做什么。

设计广度的补偿

如何做，才能认为非限制的拓宽是充分的？我们可以同时问这样两个问题，即单个抽象和领域操纵它们的接口设计问题。用东方123号作为目标系统，我们可询问应当概括到什么程度（或什么范围）：

- 对所有东方123号模型？
- 对所有双翼飞机？

- 对所有带螺旋桨的飞机？
- 对所有飞机？
- 对所有动力飞行事务？
- 对所有飞行器？
- 对所有控制系统？
- 对所有 C++软件项目？

在建立以上问题的概括的基础上，应当有比解的领域更多的问题需要解决，其中让概括点减少是由产品的经验和运作这种设计事务的性质来决定。

补偿设计的产品

我们用拓宽设计范围的方法来改进可重用，但可重用又是什么含义呢？大多数可重用的努力都集中在代码可重用上，虽然更重要的题解的可重用结构，即可重用体系结构和可重用设计。如果设计的一部分能够可重用，那么许多代码也能够重用。如果特定的抽象不适合任何现有的实现，那么类的层次结构就会提供一个建立新抽象的框架。当然，现有的设计必须完全满足这个工作；新的应用必须让现有的设计在领域范围之内使用。

可重用设计能够获取框架和库，更详细的内容将在第 11 章加以讨论。可重用问题也将在第 7 章详细加以讨论。

6.4 对象和类的关系

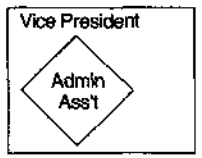
回忆一下实体之间识别关系的设计活动以及如何把这些关系转换到实现中。本节开始讨论如何拓宽面向对象设计的抽象之间关系。

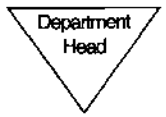
在上一节中，我们看到实体之间的关系能够直接映像到相应的 C++类和对象结构上。考虑例 6.1 和例 6.2，这两个例子描绘了实体的集合（个体公司）相互之间有某种关系。假定要设计一个反映这些抽象的软件系统（组织通信系统或者管理信息系统）。为了寻找个体公司内部组织的关系，我们就要确定类之间的关系，就要在软件中合理地表示它们。

例 6.1：在一个组织中的类型关系。

```
class Employee {
bublic:
    Employee(Name);
    ...
};

class Secretary:public Employee {
 public:
    Secretary(Name);
    ...
};

class VicePresident:public Employee {
private:
    class AdminAsst:public Employee {
```

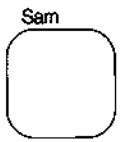
```
    public:
        ...
    } AAobj;
 public:
    VicePresident (Name me, Name Asst,
       Name sec1, Name Sec2);
        Asstobj(Asst),
        SEC1obj(sec1),
        SEC2obj(sec2),
        Employee(me)  {}
    ...
private:
    Secretary SEC1obj, SEC2obj;
};

class DeptHead:public Employee {
public:
     DeptHead(Name me, Name secy):
        SECobj (secy).
        Employee(me) {}
    ...
private:
    Secretary SECobj;
};
```

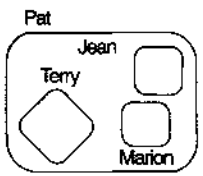


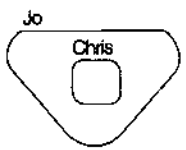


例 6.2：在一个组织中的实体关系。

```
 Secretary Sam(
    "Sam");


 VicePresident Pat(
     "Pat",
    "Terry",
    "Jean",
     "Marion");


DeptHead Jo("Jo",
    "Chris");
```







至少有一种计算机科学范畴，对定义真实世界实体之间的关系有很大的兴趣，这就是人工智能（AI）。人工智能是给彼此相关的实体有建立自己的字典的语义网络。这里，我们要利用一下这个字典，因为该字典对讨论作为反映真实世界模型的软件层次结构有用。下面我们就来讨论这样一些关系。

IS-A 关系

IS-A 关系是类型或类之间的特化关系，是一种具体-抽象关系。例如，副总裁是雇员：一般说来副总裁具有雇员的所有特性，但反过来就不能这么说了。另一种说法是，副总裁是雇员的一种特性，但不是所有雇员都是副总裁。我们可以说所有雇员的集合包含了所有副总裁的集合，所以我们可以说 Vice President IS-A Employee。当然，Secretary IS-A Employee，就好像行政管理助理（administrative assistant）IS-A 雇员（employee）和部门经理（department head）IS-A 雇员（employee）一样。

通常，我们谈论 IS-A 关系仅在类范畴之中（即，不在实例范畴之中）。这种关系总的说来对讨论特性（不是对特定特性）更有用。例如，我们常说你坐的椅子是特定椅子类型的实体，不是 IS-A 椅子。我们使用 IS-A 关系把椅子分类：作为椅子的类型，（无名的）有 5 个腿脚和扶手为红颜色转椅，有一个类型 Chair 的 IS-A 关系。

在 C++中，我们用公有继承表示 IS-A 关系。每一个这样的类都显式地从 Employee 中导出：

```
VicePresident
DeptHead
Secretary
AdminAsst
```

这就是说，Employee 的任何可访问（public）特性也是每个这样的更具体类的特性。事实上，

```
class Secretary:public Employee
```

可以读作“Secretary IS-A Employee”。

HAS-A 关系

HAS-A 内涵着包含之意，并且与 is-part-of 是可互换的。与 IS-A 不同，它是类的惟一关系，HAS-A 定义了两个类之间、类和对象之间或者两个对象之间的关系。

我们先看在特定的 VicePresident 类层上，副总裁已是一个聚集，并在公司中建立了一个新的职位，指定单独为副总裁服务的——行政管理助理（AdminAsst）。行政管理所做的每项工作完全是在副总裁指导下进行的，在管理结构上只有行政管理助理最有发言权。鉴于担任副总裁这个事实，所以这种关系出现在概括的类层上，即类 VicePresident HAS-A AdminAsst。我们可以看到，后者的类的说明在前者类的说明之内，这反映出概念上的所有（权）。

现在，副总裁没有建立秘书这一概念，但在英语的含义里很清楚，即副总裁有秘书。在例 6.1 中暗示副总裁也没有虚构部门经理（DeptHead）有秘书的概念（行政管理助理们可能有他们自己的秘书）。我们可以说副总裁的实例或部门经理 HAS-A 秘书（Secretary）。只要在类的层次上没有 HAS-A 关系，则在实例层就会明显的有一个 HAS-A 关系。类 Secretary 在它自己的权限中作为概念是独立的。在概念层上，它不被任何其他类所“拥有”。Secretary 的单个实例是与其本性有关的，并且作为从 Manager 导出的成员类来加以说明。

在实例层上的 HAS-A 很清楚它本身可以把一个对象 *B* 完全包含在对象 *A* 之内，或者对象 *A* 包含一个指向对象 *B* 的指针，无论对象 *B* 是否由对象 *A* 创建。例如，类 DeptHead 仅包含一个指向 Secretary 对象的指针（见例 6.1）。Secretary 对象的状态应当仍然认为是 DeptHead 对象状态的一部分。通过 DeptHead 和 Secretary 之间的指针仍存在着 HAS-A 关系，就好像 Secretary 对象已经嵌在

DeptHead 中一样。但是，如果使用一个指针，只有在无其他占有指向相同实例的指针的时候我们才调用这个 HAS-A 关系（即，只要一个 DeptHead 对象指向任何给定 Secretary 的实例）。

如果 C++类包含了引用，将会怎样呢？引用是对象在另外一处的别名。如果该引用用于同一类内的一个对象，那么 HAS-A 关系的存在与该引用无关，并且该引用不会改变这种关系。此外，引用说明规定了对象不可能用该类包含说明的方式完全拥有，而该对象是通过另外一个名字来存取的，这样，HAS-A 关系就可能不存在。

回过头来，试问在实例层上有 VicePresident 对象（如例 6.2 中的“pat”）HAS-A AdminAsst 对象（如例 6.2 中的 Terry）吗？回答是肯定的。此外，在例 6.1 中，“概念的所有权”关系存在于它们的类之间。在类层上查找一个已有的 HAS-A 关系（无须在对象层上存在）也是例外的。

现在我们把这个例子作一点扩充，介绍一个新的类 Manager（见例 6.3）。我们关注 VicePresident 和 Manager 之间的 IS-A 关系，以及 DeptHead 和 Manager 之间的 IS-A 关系（即 DeptHead IS-A Manager；VicePresident IS-A Manager）。那么，秘书的概念现在在什么地方用起来合适？一种结论是秘书的概念归经理（Manager）所拥有，所以我们把 Secretary 的说明放到 Manager 内。但是，如果不同的经理有不同数量的秘书，那么我们就不需要把 Secretary 实例放到 Manager 中，而宁可对具体的经理类型在每个类中说明一个或多个 Secretary 实例。

例 6.3：某些代码映像到公共结构的一部分之中。

```
class Name { Name (const char *); ... };
class Employee {
public:
   virtual char *name();
   Employee(Name);
private:
   ...
};

class Manager:public Employee {
public:
   Manager(Name n):Employee (n) {}
protected:
   class Secretary:public Employee {
   public:
      secretary(Name n): Employee (n) {}
      char *name ();
   };
};

class VicePresident:public Manager {
public:
   vicePresident (Name me, Name Asst, Name sec1, Name sec2):
      Manager(me), SEC1obj(sec1), SEC2obj(sec2),
      AsstObj(Asst) {}
private:
   Secretary SEC1obj, SEC2obj;
```

```
    class AdminAsst:public Employee {
        ...
    } AsstObj;
};

class DeptHead:public Manager {
public:
    DeptHead (Name me, Name secy):Manager(me), SECobj(secy) {}
private:
    Secretary SECobj;
};

Secretary Sam ("Sam");
VicePresident Pat("Pat","Terry","Jean","Marion");
DeptHead Jo("Jo", "Chris");
```

USES-A 关系

如果一些类的成员函数或友元函数将其他类的一些实例作为参数，那么就存在 USES-A 的关系。如果在一个类成员函数内逻辑上请一些其他类帮助的话，那么也存在着 USES-A 关系。注意，在副总裁和部门经理之间可能存在着 USES-A 关系。这个 USES-A 关系，不是 HAS-A 关系，因为由副总裁要求的部门经理的服务的引用并不蕴含着：DeptHead 对象被包含在 VicePresident 实例的说明中。DeptHead 不是作为对 VicePresident 实现的构造块来使用；但是，这两个类可通过 USES-A 关系进行协作。

CREATES-A 关系

这是在 C++中的一个例外关系。在规范格式下我们的确已经做了很多的研究，通常它是在对象和类之间的一种关系。在执行一个类的成员函数期间，一个类的实例可以要求某些其他类去创建一个实例；在第二个类上引用 new 操作符，使之执行它的构造函数。简言之，其关系就像 USES-A，但是把对象和类之间的关系用对象之间的关系替换。

在第八章，我们将使用习惯用法给出运行时更灵活的 CREATES-A 关系。使用这种习惯用法（不是 C++本身所具有的），我们将发现 CREATES-A 是两个对象之间的关系，一个对象调用另一个对象的成员函数，无须考虑类之间的关系。

把对象和类的关系放进上下文中

上述 4 种关系获取了实体和类型之间的共同的逻辑连接以及它们在实现中如何转换到类和对象中。这要依赖于应用、语言和习惯用法的使用，其他关系在用于支持开发项目的专门用语中可能是很重要的。

这些概念是非正规的，并且对这些概念不要加上太多的形式化色彩。每种需要都要在上下文中进行解释，用这种判断作为通信的辅助，而不是作为硬的和快速的设计规则。例如，Manager 和 Secretary 之间作为 USES-A 关系相互作用的特性是等同的，而实际上却是 HAS-A 关系，虽然 Secretary 有些自治，但却有着重要的相互作用，不同于它们的原则。如果我们采用此观点（真实的秘书在组织体系结构中运作）的话，那么会有更加不同的结构产生。

类和对象关系的图形表示

Booch[2] 已经开发了有关面向对象系统设计的通信结构的表示。Booch 设计的文档技术包含 4 个组件：

- 类图，它描述类结构的文档、类之间的相互关系以及其实例的通用关系。
- 对象图，表示引用对象之间相互作用以及作为参数彼此之间进行传递。
- 状态图，通过有限状态自动机来观察对象，并用成员函数调用的方式来驱动支持状态变迁的文档。
- 时间选择图，表示事件次序与系统部分交叉对象的关联。

两个最流行的表示图技术是类图和对象图，我们使用这两种图来描述类和对象的主要关系，以便能够完整地描述系统的结构。每种图都有它自己的表示和图标集：图 6.2 表示类图，图 6.3 表示对象图。

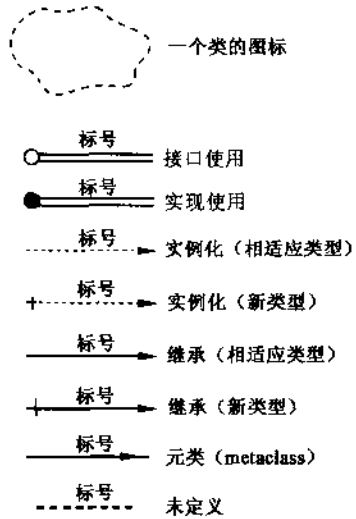


图 6.2 类关系的图标

图 6.4 表示了本章早些时候讨论过的雇员（Employee）例子的类图。该图获得了类与类之间的继承关系（用 Employee 和 Manager 作为基类），并获取了一些类层次的关系特性（每个 DeptHead 有一个 Secretary，而每个 VicePresident 有两个秘书；在 Secretary 类中的每个对象共享存取单一的拷贝机器（CopyMachine））。注意，该类图不能获取在 Manager 之内的 Secretary 的作用域。

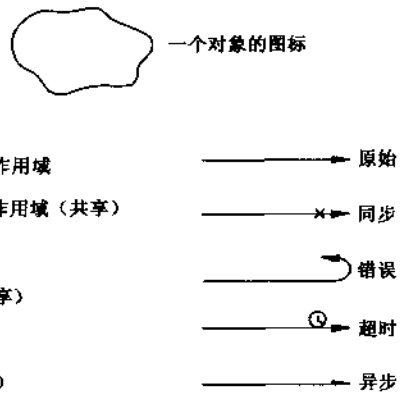


图 6.3 对象关系的图标

图 6.5 表示的是相应的对象图。这里，对象图成为 Secretary 的每个人共享其余的 Copier；Copier 是 Secretary 类的所有对象的共享域。对象图不表示 Manager 和 Employee 的实例，因为它们没有自主的实例。秘书 Jean 被表示为电话呼叫副总裁 Pat 的域。如果打电话的人要与 Pat 谈话，就要先找 Jean 预约，那么在类图中就需要更改 VicePresident 相对于 Secretary 出现在其接口中的方式：

2 Booch. Grady. "Object-Oriented Design with Applications". Redwood City, Calif.: Benjamin/Cumming, 1991.

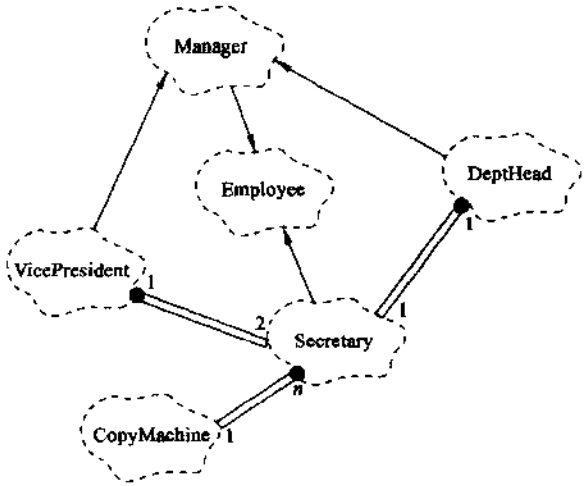


图 6.4 雇员（Employee）组织例子的类图

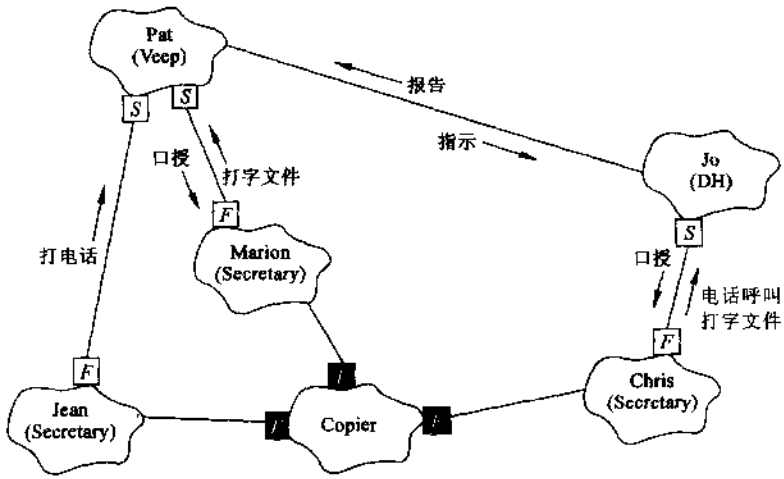
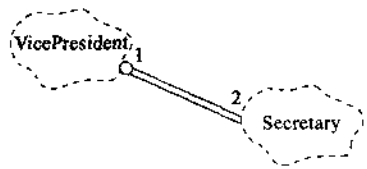


图 6.5 雇员（Employee）组织例子的对象图

有关类图和对象图的详细的描述及“面向对象分析和设计”可参阅本书的参考文献[8]。



6.5 子类型、继承和转发

以上讨论的大部分继承都与类型有关；子类型和类型经常同导出类和基类相称。这里，我们分析一下常见的共同的错误，即由在建立应用领域类型的语义和解领域的继承结构或委托结构之间划分成不合乎需要的两个类所造成的错误。首先的两 个错误是丢失“子类型”和“同构类型”，在讨

论类型和类之间关系中产生的缺陷。“类独立性”着眼于如何与封装相关的继承和委托，以及它们可能违背了数据抽象。“真正的语义成员函数”所分析的问题与“边界”和成员函数的封装有关。本书将要讨论一些特殊的习惯用法，以支持多重继承。

由于继承原因而丢失子类型的错误

一个好的工程师对实现的设计是非常关注的。在发现应用领域实体（或类型）的过程中，自然的实现策略在头脑中已经形成，包括我们可能为特定应用定制的继承结构。但这样具体的调整可能直接穿过类设计的关系，侵蚀到系统结构。这里列举一个例子以便加以说明。

再一次考虑数字计算的应用，这就是我们已建立的 Complex 类。Complex 的导出类包括 Double，BigInteger 和 Imaginary 类。Complex 高效地支持合理的操作：代数操作、有意义的计算、赋值以及打印输出等等。Imaginary 数是 Complex 复数，但 Imaginary 没有实部。

Imaginary 集合是所有 Complex 集合的子集，这一点在应用领域（数学上）的理解是很清楚的。我们认为 Imaginary 应该是 Complex 的子类型。尽管所有的 Imaginary 数是 Complex 复数的类型，但不是所有的 Complex 复数都是 Imaginary 数。

在应用领域中子类型关系指向 C++实现中的另外一个类的公有继承。我们希望找到：

```
class Complex {
public;
    ...
};
class Imaginary : public Complex {
public:
    ...
};
```

注意，这就表示了 Imaginary IS-A Complex 关系，也就是说，Imaginary 是 Complex 的子类型。广义来说，Imaginary 的集合是 Complex 集合的子集合。继承并不强制在集合中含有这些语义，也不强制 Complex 或 Imaginary 本身有这种语义。继承将增强程序员维护在层次结构中的类与类之间的兼容语义，也能够完成对相同特性的成员函数之间的兼容语义的维护。本章后面将给出维护这种兼容性的指导原则。

只要程序员遵循这些指导原则，编译就可提供强大的、安全的、方便和高效的类型系统。编译类型兼容的概念与设计子类型、相应于类的设计抽象之间的子集关系以及从另一个类来特化一个类的连接紧紧联系在一起。它把由程序设计语言表示设计关系和提供实施设计原则紧密地耦合起来。

Imaginary 真实的数据在基类 Complex 中有效。我们从在该基类的数据中计算导出类实例的有意义的数据。该导出类也携带了一些基类的外部包，如 rpart 域，此域在这里总是零。

在实施设计时，我们一直关注着实现。这里，在实现的考虑上可能暗示着在每个 Imaginary，不可能接受额外的开销，所以，Imaginary 的实现需要更改，最好采用以下这种方法，如：

```
class Imaginary {      // 没从 Complex 中派生
public:
    Imaginary (double i = 0): ipart (i) {}
    Imaginary (const Complex &c): ipart (c.ipart) {}
    Imaginary& operator = (const Complex &c) {
        ipart = c.ipart; return *this;
```

```
    }
    Imaginary operator + (const Imaginary &c) const {
        return Imaginary (ipart + c.ipart);
    }
    Imaginary operator - (const Imaginary & c) {
        return Imaginary (ipart - c.ipart);
    }
    Imaginary operator*(const Imaginary &c) {...}
    Imaginary operator/(const Imaginary &c) const {...}
    operator double () { return ipart; }
    operator Complex() {return Complex (0, ipart);}
    Imaginary operator - () const { return Imaginary (-ipart); }
    ...
private:
    double ipart;
};
```

在这个实现中有两种情况值得注意。首先，它所执行的代数操作比预处理程序的效率要高。而第二件情况更值得说一下，即 Imaginary 不再从 Complex 中继承，替代带有 Complex 的 IS-A 关系，而用 USES-A double 来代替。

两种不同体系的构造会导致不同解领域的相同抽象的优化吗？这两种不同版本的语义相同是很清楚的，但有些事情却失去了优化的意义，因为不再获取在应用领域中 Complex 和 Imaginary 之间的关系。Imaginary 仍展示了 Complex 的一些特性，并且维护了与 Complex 某种程序的兼容性：

```
Imaginary::operator Complex, 和
Imaginary::Imaginary(const Complex&)
```

这种安排具体体现了上述含义。因为不是派生出来的，所以当使用时，我们不能把 Complex 作为 Imaginary 数来进行处理。例如，转换操作符不能处理 Complex *所期望的结果，而只提供了 Imaginary *的处理。

在这个例子中需要考虑折衷方案，需要考虑类似的应用可能难以捉摸，有时甚至是很困难的。可接受的解通常只通过折衷而得到。对这个例子的解可以从 Number 抽象基类中导出 Complex 和 Imaginary，无须把任何公共数据的因子放到基类中。

但是，大多数继承的滥用都是由于经验不足造成的，这是很容易辨别的。对一个设计面向对象的新手来说，可能认为 Resistor，Capacitor 和 Inductor 类是由 Filter（称为低通滤波器类）导出的，但要说 Filter IS-A Resistor 或者 Hilter IS-A Capacitor 或 Inductor 就不正确了。因为 Inductor 的集合并不包含 Filter 集合。解决这一问题的“正确”方式是把对象的每个组成分量嵌入到 Filter 对象中，就好像一个滤波器包含了应用中的电容、电感和电阻一样，所以软件 Filter 抽象应当包含模拟的分量对象。

这里，关于另一个错误的例子是：

```
class Shape:public Point, public Color {    // no
public:
    virtual void draw ();
    virtual void rotate (const Angle&);
};
```

如果断言 Shape IS-A Color，当然是不正确的，Shape 也不是 Point。正确的手法应当是：Shape HAS-A Point 或者 Shape HAS-A Color。当然，应当这样来做：

```
class Shape {
public;
   Shape (const Point&);
   virtual void draw();
   virtual void rotate(const Angle&);
private;
   Point center;        // center和color是属性，或者
   Color color;         // 是形状(Shape)的特性
};
```

最后一个例子是 4.2 节已经介绍过的 String 和 PathName 类。如果已有的库包含了 String，我们可能诱发派生出 PathName。公有派生允许我们提供 String 期望的 PathName 对象。例如，String 的已有的 hash 函数能够可重用 PathName。但公有派生给出了 PathName 的一些成员函数，可能在一致性状态中丢失对象。PathName 构造函数和它自己的操作能够阻止在其表示中含有非法字符。但是，如果 PathName 从 String 中继承了 operator[](int)，那么下面的代码就违背了限制 PathName 的自己操作的实施的原则：

```
int main () {
    PathName batFile = "AUTOEXEC.BAT";
    batFile [0] = '*'; // 非法文件名字符
    ...
}
```

正如 4.2 节所言，私有派生能够巧妙地解决这些问题。

有问题的继承结构可以用建立派生的变异方法来解决。变异方法可以基于 IS-LIKE-A 关系在另一个类上构建一个类，并用 IS-A 关系来代替。变异最接近的关系是逆继承，逆继承有它自己的可重用工具。例如，一个图形包可以包含 Shape 类，也可以导出 Circle 类。所需要的椭圆，我们可以通过继承概括出 Circle，重写其操作来产生一个 Ellipse 类。这些都是带有风险的方法。

伴随继承：同构的类型

我们已经着眼于应用抽象的语义行为，并且用表示这些语义行为的类来导出系统结构。类行为应定义内聚性的抽象，并且在相同的继承层次结构中的类用它们提供的服务来进行组合（聚集）。行为的组织也是类似的，但不等价，而是用外部特征来组织类。这两种范型（模型组织）之间的差别，首先是要求对该类的语义有深度的理解；第二是能够用词法名字进行比较。第三种组织类的方式是基于它们的内部来表示结构。除了是在类的内部，而不是在它的接口上之外，这类似于第二种方式。在图形程序中的 Line 类包含两个 Point 对象，所以构成了 Rectangle。我们可能得出这样的结论（这是不正确的）：一个类应当被另一个类特化。

当设计类接口或用继承把类分组时，最先考虑保持行为特性是重要的。在类继承层次结构中 Liskov[3] 定义了合理安排类的准则：

置换特性的方法是：如果类型 S 的每个对象 o_1 是类型 T 的一个对象 o_2（这样对所有程序 P 根

3 Liskov，Barbara．"Data Abstraction and Hierachy." SIGPLAN Notices 23,5 (May 1988). p.25.

据类型 T 来定义）的话，当 o_1 由 o_2 置换时，P 的行为不改变，那么 S 就是 T 的子类型。

例如，程序 P 感兴趣的一些函数，当对作为参数传递时的 Complex 类型（类型 T）的对象（o_2）编写操作时，应当能够接受作为参数 Imaginary 类型（类型 S）的对象（o_1）。在实践中，把 T 成员函数应用到 S 对象是安全的，因为当 S 对象（o_1）被 T 对象（o_2）置换时，所有 P_s 是不改变的。看来，这样做是合理的，因为 Imaginary 可以要求 Complex 做能够做的任何事情。注意，逆命题是不正确的：对 Imaginary 操作的函数设计通常不可能接受 Complex 类型的对象。如果试图存取任意 Complex 内部的 baseNameVal 成员，那将会发生什么情况呢？

这个准则（称为 Liskov 置换准则）对成功地进行面向对象设计是相当重要的。

某些设计者认为使用继承无须遵照置换准则。有时，这些使用作为子类型实现比用设计方法实现更有效。但这样的应用会导致对程序设计语言同构模拟的误解。如果滥用这些作法会导致对象范型的“回退”，例如，从 Circle 派生出 Ellipse，把 Circle 修饰概括到 Ellipse 中。因为这种情况能够清楚地标识，而对 Liskov 置换准则的应用产生怀疑，在这里我们不再详细研究。大多数的滥用将陷入到变异的构造中，试图使用 IS-LIKE-A 关系而不是 IS-A 关系来“侧向继承”。

用集合（Set）和链表（List）作为启发性的例子

考虑已有的类 List，它作为对象收集有序的容器服务。List 并没有假定包含了该对象的特性，即它可能是作为 void*对象的链表实现。List 可能有这样的表示：

```
class List {
public:
   void*  head();              // 返回到链表的头
   void*  tail();              // 返回到链表的尾
   int    count();             // 返回链表元素计数
   Bool   has (void*);         // 对链表中元素成员进行检测
   void   insert (void*);      // 对该链表增加一个新成员
};
```

现在我们要构造一个新类 Set，可以这样设计：

```
class Set {
public:
   int    count ();            // 返回集合元素计数
   Bool   has (void*);         // 对集合中元素成员进行检测
   void   insert (void*);      // 对该集合增加一个新成员
};
```

我们很容易发现这两者之间的相似性；我们甚至可能找到工具以证明 List 类的存在，并且能够作为实现 Set 的基础。所以我们试图使用相应的继承：

```
class Set:public List {
public:
   void insert (void *m) {
      if (!has (m)) List::insert (m)
   }
private:
   // 下面两行是非法的 C++构造
   List::head;              // 集合的头未定义
```

```
    List::tail;              // 集合的尾末定义
};
```

即，为了实现集合的语义（相同的值要在集合中多次插入）就要考虑冗余问题，我们必须重新定义insert(void*)成员函数。这里要继承Count()和has(void*)操作符，我们的意图是使head()和tail()难以达到，因为它们未对Set定义，虽然它们有List的含义。

但是，这里我们正在建立一个重要的问题。因为是派生，所以我们可以说Set IS-A List。换言之，我们断言List对象的任何地方都能用Set对象替换。这当然是不对的，反之也不对。就这点而言，应注意它们各自假定对同一值的多次插入的处理：对Set插入是个空操作，对List不是。此外，Set对head()和tail()的操作不处理语义，C++提供后者情况的指引：以上Set的说明，其中List::head和List::tail是作私有而重新定义的，在语言中这样定义是非法的。如果要合法的话，那么就有含糊的语义，特别地，当这样的函数是虚拟的时候。

List和Set几乎是**同构类型**：它们有相同的特征但却有不同的语义含义（如果List不实施head()和tail()的话，那么其类型最好是同构的）。这就产生了在继承层次体系结构中抽象特征的Liskov准则，正如本节开头所说的那样。

让我们再来看看另一个例子，它是Sethi[4]在“程序设计语言”一书中提到过的。如果已存在CircularList类，并且我们要建立一个Queue类，则我们可以写成：

```
class CircularList {
public:
    int    empty();
           CircularList ();
    void   push(int);          // 压在头部
    int    pop ();
    void   enter (int);        // 压在尾部
private:
    cell *rear;
};
class Queue:public CircularList {
public:
           Queue () {}
    void   enterq (int x)      { enter(x); }
    int    leaveq ()           { return pop(); }
};
```

注意，Queue没有理由得到push(int)和pop()操作，派生的副作用对Queue是毫无意义的。这个问题的根源是由于使用了公有派生。一个较好的经验是公有派生应当使用子类型关系。如果该派生是方便的或可重用的，那么就应当使用私有派生。私有继承（优于替换）将在第7章中加以介绍，作为代码可重用的手段。

当继承有在一个类型中表示另一个类型的可重用目的时，问题相似（或许是糟糕）集合就产生了。例如，有一种可能是，从List派生出Set，其内含是Set的内部会同List内部相同，那么可能把这些操作理解成有一点点不同。这样可以明显地看出其与类型和子类型模型无关，并且应当避免有关联。

4 Sethi，R. “Programming Languages”. Reading，Mass.：Addison-Wesley，1989.

无伴随继承的解

当继承似乎像是一个很好的解时，通常用更直接的机制就可以了。例如，重新考虑本节的 List 和 Set 的例子。你可能会很失望，因为许多事情并未设计好。确实存在着用 List 这样的一些操作的可重用良机去实现 Set，它正好因为采用公有继承这种错误的作法而把这种机遇丢弃掉。这里我们采用另外一种方式：

```
class List {
public:
    void*     head();          // 返回链表的头
    void*     tail();          // 返回链表的尾
    int       count();         // 返回链表元素计数
    Bool      has(void*);      // 检测链表中的成员
    void      insert(void*);   // 增加链表中一个新成员
};
class Set {
public:
    int       count()          { return list.count(); }
    Bool      has(void*m)      { return list.has(m); }
    void      insert(void*m)   { if (!has(m)) list.insert(m); }
private:
    List      list;
};
```

我们需要显式地对内部列表“转发”Set 的 3 种操作。注意，我们对无效的 head()和 tail()什么也不做。

这种方法的优点是可重用——List 在功能上支持 Set 的语义。虽然在 List 和 Set 之间有一些相似性，为此我们放弃对它们的类型之间的关系有意义的要求。这种方法的实现是指向设计的，对 Set 和 List 之间的关系没有说明。这种实现使人联想到委托，委托不是基于类的而是基于实例。C++支持从一个类到另一个类的操作的转发，能够作为委托的一种弱形式。这已在 5.5 节中讨论过。

带有添加和重写功能的继承

按照 Liskov 的置换方法准则，为了实现子类型而合理地使用继承，就可以使用带添加的继承方法来完成。添加类的成员函数直觉上可以减少对象个数。例如，对电话添加快速呼叫函数，创建一个与普通电话相比具有更多功能的新的电话抽象：快速呼叫电话的集合是所有电话集合的子集。因为已有的操作被完整地保留下来，导出类的对象仍能使用所期望的基类对象，导出类的不必要的操作在此不作讨论。但是，这也意味着软件可以操纵这样的对象，根据它们的基类接口将不能把新的成员函数添加在低层上。换言之，SimplePhone 和 SpeedCallListPhone 之间的差异迫使程序员处理它自己的每个项。对采用 SimplePhone 作为参数的函数说明将接受 SpeedCallListPhone，但它不能够执行快速呼叫这个特征。

使用带添加的继承，我们能在导出类上添加新的成员函数，但不能根据它们的基类的接口访问这些函数。这是继承十分受限的模型，并且在运行时没有为迅速调遣新函数留下足够的空间，所以对象范型大部分的能力不能展现出来。如果程序员在继承层次结构中必须处理每个类的它自己的项的话，那么继承就不提供抽象能力。程序员应该能够处理类的巨大的聚集，处理所有这些聚集就像处理一些类的实例共享它们的公共基类那样。

另一个极端是抽象基类的使用（纯特征的类），它缺乏应用的语义。缺少语义，我们能够确保与导出类没有语义冲突。例如，“宇宙基类”可以有操作 at，atPut，more，moreNow 和 getNext，通过它的作用域来访问对象，或者作为连续的聚集加以处理。at 操作可以用串或枚举值作为键惟一地标明一些作为返回的内部值（换句话说，亦即相关检索）。atPut 成员函数可以采用一个键和一个值作为参数，其中键指定了存储该值的那个域（相关存储）。此外，一些 atPut 操作可能被判定具有某些副作用，其中特殊的键触发了内部成员函数的执行。下面，我们来考虑一个相关数组的例子：

```
class AssociativeArray {
public:
   void atPut(void *element, String key);
   void  *at(const String&);
   ...
};
int main() {
   AssociativeArray a;
   a.atPut((void*)233, "AssociativeArray example");
   a.atPut((void*)230, "CircularList example");
   int circlistPage = a.at("CircularList example");
                                        // 应当返回 230
   a.atPut((void*)10, "!size");         // 改变数组的大小
   a.atPut((void*)0, "!print");         // 打印这个数组
   int size = a.at("!size");            // 得到数组的大小(尺寸)
   ...
}
```

这里，我们使用相关数组来存储书中标题的页号，用链来检索。但是，也有用特殊键“!”来指示 at 和 atPut 的，但这样做具有一定的副作用：打印、修改和检索该数组的长度，以及可能的其他要求。

这种程序设计风格通常称为基于框架的程序设计，也是在认识科学中盛行的风格，习惯上用于许多符号语言的上下文中。其结果会造成在基类中出现“胖接口”，使许多成员函数以及任意无关的导出类能够联合在一起。这种程序设计风格将在第 8 章中加以介绍。

这样的接口几乎是无限灵活的。但是，它几乎不通信或者不执行设计的意图。这种实现执行起来很可能非常糟糕，因为它要求一个串能查阅每种操作。我们要求类是一种恰当的媒体，接口一般语义的通信无须要求任何特殊的实现。这样，接口能够表述在基类中的特征，此特征为导出类中后来的实现定义服务。在继承层次结构顶部的基类可能从来不会建立对象，但它的导出类对象习惯作为编译时的接口之用。基类表示了在基类下面的所有类的公共特征。例如，我们有一基类 Window，说明本身有操作 move(Point)，addChar(char)，clear()和 delete Line()等。Window 导出类包括 CursesWindow，XWindow，MSWindow 和 SunViewWindow。我们从不在 Window 本身实例化 Window，因为它不知道如何做这种事情。我们仅用实例化导出类来代替。具有这种特性的基类称为**抽象基类**。有关 C++的抽象基类的更详细的描述参见 5.4 节。对窗口的集合，其框架可以如下编写：

```
class Window {
public:
   virtual void  addChar(char)    = 0;
   virtual void  clear()          = 0;
```

```
    virtual void  deleteLine()     = 0;
};
class CursesWindow: public Window {
public:
    void  addChar(char c)     { /* 合适的算法 */ }
    void  clear()             { ... }
    void  deleteLine()        { ... }
};
class XWindow: public Window {
public:
    void  addChar(char c)     { /* 合适的算法 */ }
    void  clear()             { ... }
    void  deleteLine()        { ... }
};
//etc.
```

类 Window 的所有成员函数都是虚拟的，并且没有数据。这样的类称为**纯抽象基类**——它们具有抽象数据类型特征，但有关实现方面的内容却什么也没说。

在导出类中重新定义基类成员称为**重写继承**。当基类成员函数在导出类被重写时，成员函数的语义应当被保存。我们根据虚拟函数周围的上下文，在入口和返回处编写一个虚拟函数[5]。这个上下文是当前对象的状态，或许用一些外部状态信息来增加。在入口处，基类虚拟函数应当假定它的上下文不比它的导出类副本少，而返回处应当假定不比它的导出类副本多。当基类成员函数在导出类对象上被引用时，入口准则是基于上下文的变化制定的。在这种情况下，应当在导出类成员函数中处理调用。因为导出类成员函数必须能够处理所有基类版本的要求，所以可以假定只要基类版本执行就行了。对出口准则的补充是必要的：导出类成员函数必须至少完成基类版本所期望完成的工作，虽然导出类成员函数做得可能更多。

如果符合这些准则的话，那么导出类成员函数是与基类中的一种在语义上兼容。这些准则在应用领域中是清楚的，其中类型之间的关系有形式化的基础，例如，在算术类型的层次结构中（实数、有理数、非负数、基本（数）等）。建立这样的准则在图形领域、商业系统、电信通讯和其他很多领域中需要再多做一些工作。

注意，在抽象基类中，所有纯虚拟函数一定满足这些准则。所有有关在引用中的上下文的假定都是传送一个给定的某个类型的参数个数。纯虚拟函数没有代码，因此不做修改上下文的工作，而且成员函数返回类型是所有存放完整上下文的声明。

在基类（非纯）虚拟函数的公共情况下，在导出类中重写，两个函数至少有一些语义上的不同，如果它们相同，则基类函数能够被直接使用。但是，两个函数应当像上面描述的那样，应在语义上兼容。设计者必须确保导出类成员函数有相同的语义。

在使用继承的时候，可能有两种极端。首先是在抽象基类中，此时在层次结构中的所有类的特征是相同的，但新的导出类在它们父辈操作的语义上有难以捉摸的变化。另一种极端是，带添加的继承（留下了未触及的实现）改变了严格向上兼容方式的接口。遗憾的是，大部分继承的公共使用两种极端情况都没有介入。抽象基类没有数据，而纯虚拟函数却是不经常发生的。设计一个好的抽

5 参见 Meyer，Bertrand. "Object-Oriented Software Construction." Englewood Cliffs，N.J.：Prentice Hall，1988. 257.

象基类应能反映它们的导出类的公有行为。这种设计通常首先考虑的是可重用：公有成员函数和实现数据是纳入基类的重要因素。这些公共基类行为通常是缺省的，需要在导出类中重写。如果对保护成员函数语义特别关注，即从基类到导出类的所有行为，那么我们就完成了继承层次结构的体系结构抽象。

另一方面，带添加的继承也存在一定的问题。如果应用分析表明一些函数只属于导出类，并且不在它的基类中，那么就会失去一些体系结构的抽象。例如，考虑一个能够填充和不能填充形状的Shape库，如图6.6所示。如果我们添加fill成员函数使形状内部填充了颜色，则应当如何进行？如果把此函数放进Shape中，我们就有一个建立纯虚拟函数的选择，或者给出fill函数的一些缺省行为。构建的纯虚拟函数迫使fill成员函数定义Line和其他开放形状，有些开放形状是没有意义的。缺省行为可在Shape::fill中建立，但对Circle有意义的缺省对Line以及它的同类可能是没有意义的，所以两者的含义不能都正确。作为虚拟函数定义的fill在CloseShape（封闭形状）内，在语义上大部分是合理的，但它的应用方法一定要填充的对象现在必须看到CloseShape类的接口，就像看到Shape类一样。如果CloseShape和OpenShape（开放形状）原先不在继承层次结构中（可能给出的fill未考虑在原始设计中），那么寻找对fill成员函数语义合适的空间就会更加困难。在进退两难的情况下通常做什么还能保持一点语义呢？在Shape中定义fill作为具有空主体的虚拟函数，更严格地选择可能会给Shape::fill带来一个例外。

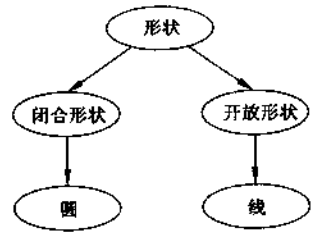


图6.6 另一种形状库

大多数真实的情况是落入了纯添继承的“规则”极端和抽象基类之间，因此这是受捕捉子类型陷阱的支配的。在本节开始介绍过的例子中，Set是从重写insert(void *)成员函数的List中导出，但可能恰好就是这个原因而产生失效。这样混合的继承在程序中经常发生，会对维护和解的演化产生难以理解的困惑。这个问题无论是在论文还是在实践中都是很清晰的，但是基于继承的程序设计而不是子类型会维护该问题。多态性会进一步使该问题恶化。

注意，该问题只有当使用了虚拟函数时才会发生，所以相关的类能够使用多态性，让我们来看看使用重写继承的Set和List的实现，它们是作为两个完全不相关的类而被处理的。即，List按照List方式处理，Set按照Set方式处理，而且并不企图一个替换另一个，那么重写的出现就不会产生病态的副作用了；这是一种与应用设计特性不相关的模拟解。这些抽象之间子类型的不确定以及所隐含着的一种抽象与另一种抽象互换的能力，从未发生过应用空间问题。但是，如果List函数是虚拟的，并且使用这种虚拟，那么就存在着显式使用继承重写特性，并且不确定现象就会发生。继承（能够反映类似的或应用子类型的实现）和多态性（解领域构造习惯用于支持应用领域子类型）之间紧密耦合使其结果产生真正的问题。

我们再假定List和Set是作为独立的对象而被处理的，在这种情况下来处理是不错的。在前面“无伴随继承解”一节中包含的解是一个未使用继承的Set和List组合的例子。而Set对象向嵌入在它里面的List对象转发要求，两种类型紧密配合，一个向另一个转发一些操作，很少以建立子类型的设想为根据。这里还有一些例外，特别是在C++程序设计的高级风格中，本书后面会加以讨论。但通常情况下，转发能够支持多态性或者多态性的外部特征，无须牵连到继承。转发的下降是比继承在编译时间类型安全更少，而且比继承能够捕捉的代码设计信息更少。

带注销的继承

一些面向对象程序设计的环境允许在导出类中“减去”或取消基类特性。C++不允许在公有继承中注销。编译时取消的执行与导出类对象可能使它的基类对象在运行时出现转换冲突。这可以通过以下片段代码来看出：

```
class Base {
public:
   void function1();
   void function2();
};
class Derived:public Base {
private:
   Base::function1;                // 取消(在 C++中无效)
public:
   void function3();
};
void f (Base *baseArg) {
   baseArg ->function1 ();         // 出乎意料:忽略取消
}
int main() {
    Derived *d = new Derived;
    f(d);
    return 0;
}
```

如果取消是合法的，编译需要执行难以处理的完整的数据流分析，以确保可以永远使用对象的特性而不被注销掉。如果该函数是虚拟的并且是分开编译的，则这个问题就变得更加严峻。一个对导出类说明访问的模块创建了一个导出类对象，这个对象用基类指针指向另一模块中的一个函数，这另一个模块没有访问这个导出类说明，并且也不知道注销情况。这样，第二个模块就可能引用对象上的基类成员函数，而这些函数已经被注销掉了！例如，考虑在上述代码中已经发生过的情况，如果 function 1 和 function 2 是虚拟的，而 *f* 已经在各自的源文件中编译了，仅有 Base 说明为可见的。

运行时执行注销会使语言的实现过于复杂化，并要求每个类对象携带更多的软件包，以便指出这些没有被伪装或冒充的取消过的成员函数。

基类数据能够使导出类及其用户变成不能被访问，但它们却能被取消掉（基于节约导出类对象内存的原因）。对虚拟函数而言，成员函数引用在编译时传递类型系统是确保运行时调用什么信息，而不是给出不了解信息的因素，就像在 Smalltalk 语言中那样。如果导出类提供没有相应名字的函数，那么某些父类的同名和同特征函数将被使用。

无多态性派生稍微有点不同。正如我们前面看到的，编译时通过重新拷贝来取消是不允许用语言的，例如，

```
class List {
public:
    void *head ();
    void *tail ();
    int count ();
    long has (void*);
```

```
    void insert (void*);
};
class Set:public List {
public:
    void insert (void *m);
private:
    List::count;            // 错误
    List::has;              // 错误
};
```

是非法的。但是，如果私有派生被执行——高效地取消所有该基类的公有操作，这个基类就能够一个接一个地被访问：

```
class Set:private List {
public:
    void insert(void *m);
    List::count;
    List::has;
};
```

带有取消的派生可能暗示着不合适的继承结构已经存放在适当位置。例如，在它们的父/子关系中具有派生反向或侧向关系。一般地，问题充满在注销领域中，并且应当做为设计以避免它，甚至作为设计意外事故而需倍加关注。

需要“真实语义”的成员函数

在面向对象设计和程序设计中，在对象上的操作相对于真实世界中的某些语义。这是比所有对象用定义好的接口访问还要强和深刻的描述。维护这种语义标识对成功地设计是非常重要的。封装不是作为存放与可见性操作相对的数据的许多特性的仓库，而是如何组织这些行为特性的地方。

我们用两个例子来强调说明这个准则。这两个例子十分关注封装和继承之间的相互影响。

例子1：get 和 set 操作。

程序员经常采用“公有数据是有害的；数据应当总是通过 get 和 set 操作来封装和访问”这样一种面向对象的指导准则。一个是根据内部数据表述的问题，特别是解领域概念的表述。另一个问题是有关对象整体行为，或者是连带的有关应用领域类型。即有关该对象作为整体或者它的行为特征被标记上成员函数，这是一种技巧上的处理。

成员函数的名字应当是生动的，语义上是丰富的，并且在类的上下文中具有清晰的含义和解释（要见名知意）。get 和 set 成员函数的出现在设计中应当重新考虑用红色来标记。在重新查找内部值时对象的活动或职责是什么？用户通过设定内部状态值想得到什么？这些问题的解决像是辅助这些与应用相关的类成员函数，并且帮助它们分离出什么样的类需要从用户了解到相应的可见性要求。成员函数仅作为调节内部数据服务，并不保持在 C++作用域规则内的类的封装性；反而是在 C++作用域内破坏了封装性。

例子2：公有数据。

当然，使类的数据公有违背了封装性。在另一个对象 *A* 的公有接口中成员对象 *B* 的外部特征使具有 *B* 行为的 *A* 接口造成了混乱。*B* 的行为根据 *A* 的响应度来表述是不太可能的。*B* 应当通过 *A* 的成员函数来操纵。这就是说，应当“统一”和“平滑”地舍去 *A* 接口，无须用户提供更深层次的要求：

不正确的编写：

```
class window {
public:
    cursor cur;
    ...
};


int main() {
    window win;
    win.cur.move(10,5);
    ...
}
```

正确的编写：

```
class window {
public:
    void moveCursor(int,int);
    ...
private:
    cursor cur;
};
int main () {
    window win;
    win.moveCursor(10,5);
    ...
}
```

这里，win.moveCurser 是 win.cur.move 的生动的表达式，因为它能够根据一种抽象而不是两种抽象来理解。

例 6.5 在继承下变得更具趣味性。Snyder[6] 提出大多数面向对象语言允许导出类访问基类的内部数据，此外，数据不是类公有生成的。在 C++中却不是这种情况。C++用户能够获得更流行的语义描述（采用 Snyder 给出的数据域的 protected 属性）。

继承性和类的独立性

面向对象设计是以数据抽象为基础的。数据抽象、概括技术，通常是与封装信息隐藏技术联合操作的。即一旦数据抽象试图隐藏其细节之时，如果你希望的话，则不是在本身来防止寻找内部数据。另一方面，封装强烈地抵抗撬开接口窥视其细节。抽象与应用领域有关，而封装与解领域有关。

数据抽象和信息隐藏对面向对象设计来说，既不必要也不充分，但它们是很好的面向对象实践的重要组件，在面向对象程序设计中也具有很强的独立性。进而，它们的趣味性是根据是否继承性违背了数据抽象的不断进展而得到发展的。

这里有我们希望保持的 3 个数据隐藏特性，并且成为是否违反封装性或信息隐藏的试金石。我们随着变化对它们加以描述；这些技术对程序的演变和发展是很重要的。用“改变”方法，就意味着需要编辑更多的抽象类型内部表示或特性函数的传统模式。改变也必须由继承引起，其中不同的改变能够把一个类应用到所定义的一个新类中。

（1）对一个类要求在其他类中重新考虑设计决策会做改变吗，或者它们是完全的包含吗？

（2）在一个类上做一些操作（例如，内部编辑，或继承和转发的外部应用）会有类中的语义或行为上的潜在变化吗（一种方式是根据变化需要而做出重新测试）？

（3）若要改变一些类应该作出怎样的努力（例如，重编译的花费）？类的大小，类的合作的大小，使用类的程序大小，或者包含类的项目的大小是成比例而协调的吗？

这里，我们仅从继承的观点来检查抽象和数据隐藏，即，是否继承建立了特定问题或提供了特殊的机会。

C++允许控制基类成员（数据或函数）对导出类的存取能力，所以它是 C++程序员实现了强制数据隐藏和封装的语义。公共的公有派生有合理的语义保持封装性：

6 Snyder, Alan."Encapsulation and Inheritance in Object-Oriented Programming Languages." SIGPLAN Notices 21,11 (November 1986).

- 基类私有成员不能直接访问导出类；
- 基类公有成员访问导出类恰好与它们访问其他代码一样；
- 基类的公有成员变成了导出类特性（公有接口）的一部分。

但是，还留下了一个需要认真对待的问题。虽然C++语言的抽象构造限制了私有数据变化的语义作用域，但重编译的影响仍然是严重的。下面是大多数C++环境中相互作用的一些应用：

- 改变基类的表示要求对所有它的导出类重新进行代码编译。
- 添加基类虚拟函数要求进行类似的重新编译。
- 添加基类非虚拟函数可能不要求重新编译导出类代码。但是，如果新的函数隐藏在同名（在它的基类中）的另一个函数中，那么所有代码引用那个名字的函数需要重新编译。

在C++中，另一种非法手段允许破坏封装性。让我们假定类Shape使它的center域在保护节（protected），即，center能够访问“内部”（导出类），但不能够访问“外部”：

```
class Shape {
public:
    ...
    virtual void draw () = 0;
    virtual void move (Point) = 0;
protected:
    Point center;
};
```

现在，类Ellipse可从Shape（形状）中继承并操纵center。这里存在着有意识地违反封装性的问题，但似乎有点不值得，特别是在如果Shape是个抽象类的情况下。

```
class Ellipse:public Shape {
public:
    ...
    virtual draw();
};
Ellipse anEllipse;
```

现在我们能够操纵anEllipse，但我们需要通过语言用直接操纵center来防止。这里有一种欺骗方式：

```
class Cheat:public Ellipse {
public:
    void cheat (Point p) { center = p; }
};
Cheat anEllipse;
```

Cheat从Ellipse派生出来违背了应从Shape中抽象而不是从Ellipse中抽象的原则。这是逆直觉的，并且违背了“最小意外的原则”：在添加Cheat的某个地方都没有产生Shape问题。该变化也违背了Shape的封装性。作为一个更具隐蔽性的例子，考虑下面的代码：

```
Point center;          // 整个图形的中心
class Circle:public Ellipse {
public:
    void incidentalCenterReference () {
        ... center ...
```

```
    }
};
circle aCircle;
```

假设 Circle 的创造者期望引用 center 对 Circle 对象的 Shape::center 域加以判定。如果 Shape 从未实现 center 域的删除、重命名和私有化，那么同一个引用将与全局 center 捆绑在一起；编译时没有任何错误，但却有错误的语义。在这里继承违背了抽象，但这仅仅是因为它对某些类成员提供了 Shape 的 protected 状态。因此需要经常有意识地对性能调节、内存优化和代码可重用作综合平衡。有关 protected 域和可重用的问题将在第 7 章继续进行研究。

一般来说，虽然 C++类保持与作用域构造的语言学上的独立性，但语言的语义使得 C++类几乎不太可能消除变化所产生的广泛的影响。对小的程序来说，完成重新编译是可以接受的，这不是什么问题。但对大型应用程序来说，因效益的原因，重编译受到了限制，理想的代码发生了改变，某些程序设计风格和习惯用法的应用可以缩小这种影响。这样的习惯用法在第 5 章到第 8 章均作了描述。

6.6 子类型、继承和独立性的经验规则

在继承层次结构中导出类函数重写了基类虚拟函数，这是给定对象的能力，但必须小心使用。这样的派生应具有以下一些限制：

- 在基类和导出类之间存在着形式上或强直觉的子类型情况，其中重写与用导出类表示的基类对象子集边界上的限制有关。好的但不是理想的例子是从 Ellipse 导出 Circle，依靠重写 rotate(Angle)操作而实施（尽管这不是理想的例子，因为从纯类型观点出发，由椭圆定义的 rotate(Angle)操作，对 Circle 执行得很完美，并且没有重定义的事务，但对我们的设计并不理想）。
- 在基类语义与这些导出类兼容的情况。在入口处，基类虚拟函数应与假定有关它的上下文相比，其导出类副本不会少；而在返回处它应与有关上下文相比，其导出类副本不会多。在抽象类中，纯虚拟函数确实满足这些规则。
- 实际上，若真的使用了带添加的继承，并且该函数说明为方便起见放在了基类接口中。子类可能用它自己特有的操作作注释，就安全和语义两方面来说，在兄弟类（类共享基类）实现这些操作，包括作为无操作的实现，那么该函数能够在空主体的公共父类中用虚拟函数来添加。例如，具有导出类 ClosedShape 和 OpenShape 的类 Shape 可能有一个 fill()操作。基类作为无操作来实现，并且这些语义通过形状（填充是一种蜕化情况）来继承，仅用 ClosedShape 导出类来重写它。这样的基类称为胖接口。实际应用时应当适度。
- 在导出类和基类没有 IS-LIKE-A 关系的情况下，它们应当具有 IS-A 关系。如果发现了两个类具有 IS-LIKE-A 关系，那么查找或创建这两个类的一个公共基类，可作为它们的共享的父类。

我们也可以明确表达有关在 C++中数据隐藏的一些经验规则：

- C++要求，只有在类的输出符号特征改变时，设计才根据在类内的变化而进行修订。即，类型用它的特征（公有接口）来特性化，并且通过 C++编译强制使用和类的定义之间保持特征的一致性。

- 告知在协作类中类的无效设计决策语义是否变化是困难的，即使在接口保持不变也是如此。但是，程序员通常在这个领域中的直觉是不错的，并且大部分好的行为维护了非公有数据结构和算法的封装。
- 用避免保护数据（protected data）方式确保基类的封装性：保护数据从导出类观点来看有点像公有数据。但是，在应用中总是使用保护数据之外的 friend 或 public 数据。
- 一般地，改变一个类的实现要求对类的代码重新编译，这个类是从改变的类中导出或者依赖于它的接口。在系统支持把 C++代码递增地装入到运行程序中（作为连续运行嵌入系统）时，任何对类的实现的改变将可能要求所有这些类的对象在不同程度上加以更新或替代。

另一个有价值的经验规则是，一般地，放在库中的类成员函数应当说明为是虚拟的，即使在这些成员函数的初始应用中没有导出类。当发现新的类和该库的类之间具有一定关系的子类型时，就允许以后的继承。虚拟函数必须被 C++类所使用，这要得益于面向对象设计技术。

一个可能类似的经验是建议所有导出的类要作为虚拟派生，以支持在将来的多重继承的潜在的演变。但是，通常这种决策会给设计者带来新的子类，并且不能总是由父类参与。在某种意义中，父类的虚拟性应当根据它们的子孙的意思而决定。然而，C++强制这种决策是在派生的层次结构的高层上加以实施。这也是语言研究中的一个高级课题。

练　习

6.1 指出 IS-A，HAS-A，USES-A 中哪一种关系有以下特性：

（1）传递性（transitivity）

（2）关联性（associativity）

（3）可交换性（commutativity）

6.2 考虑下面的例子：

```
class CopyMachine {
public:
    ...
};
class Secretary:public Employee {
public:
    Secretary (const char *n): Employee (n) {}
private:
    static CopyMachine copyMachine;
};
Secretary Connie ("Connie"), Terri ("Terri");
```

（1）Secretary 和 CopyMachine 之间的关系是什么？

（2）如果类 Manager HAS-A 类 Secretary，并且类 Secretary HAS-A CopyMachine 对象，那么 Manager 和 CopyMachine 又是什么关系？

6.3 对本章所介绍的 Account 例子编写出实现代码。

6.4 仅用成员函数是 at，atPut 实现关联数组，并且这些成员函数要符合传统标准格式的。

6.5 使用枚举值（enum value）作为键重新实现关联数组，并比较这两种关联数组的性能。

第7章

重用和对象

重用（reuse）是一个避免做额外重复工作的概念。我们可以重用某种概念，可以重用设计和主要系统接口，也可以重用源代码和对象代码。本章讨论如何把对象范型和在特定的C++中帮助开发者构造可重用软件。

软件重用是个大课题，对其要进行完全的处理则超出了本书的范围。虽然面向对象设计和面向对象程序设计促进了重用的发展，但我们必须强调重用是软件工程的规范，是一种方法学和设计规则的技巧。真正的重用是预先思考，计划好的，并且能够确保仅用合适的设计技术、文档、软件管理和分布机制就可以达到。重用应当与软件利用（software salvaging）相对照，在已有的系统中发现代码的逆向工程实践恰好适合这种目的；重用也应当有别于软件存储（software carryover），软件存储是当软件中的有意义的一个程序片段嵌入到系统中并为多个系统服务时发生的。软件利用和软件存储都是有价值的，但不能获得像运行良好的重用程序那样具有相同的优势和效益。

本章集中在C++的语言构造和习惯用法上，并且仅着眼于在设计上的考虑。本书的许多章节已经涉及到重用，6.3节也讨论了重用设计的重要方面——领域分析。

代码重用是解领域问题，继承是重用的工具。基类在功能上提供了导出类，无须添加额外的代码。面向对象重用为软件带来了较高的生产效率，并且可以合情合理地获得，这些在实践中得到了证实。

面向对象设计有时与在对象范型下的重用设计会产生混淆。一种观点认为，如果类结构优化了重用，那么最有可能使面向对象设计获得最大效率的是使用继承；另一种观点认为，受应用领域支配的正常的类型关系将自然而然地减少代码的冗余程度。但这两种看法都是错误的。

我们用寻找相似实体的方法来标识重用，试图通过继承来刻画重用代码。但回忆一下继承是解领域的概念还是很有必要的，因为它同时使用重用并反映子类型的思维。为了确定如何反映公共代码的重用，我们需要了解对象之间的关系：它们是存在IS-A关系呢，还是存在IS-LIKE-A关系？如果是IS-A关系，那么我们应当寻找设计重用，而不是代码重用。如果是IS-LIKE-A关系，那么就潜在地存在着代码重用。但是，应当注意，必须不采用具有公有继承表达的IS-LIKE-A关系的重用。在6.5节中我们看到，利用某些重用方式总想从List中继承Set。我们能够实现某些重用，但当我们尝试在基类上下文中使用导出类对象时却遭到失败。这种对象的“插接兼容”是集中在面向对象设计中，并且它的关系伴随着重用。通常，使用公有派生的决策仍应当采用第6章的考虑来驱动。

让我们考虑有IS-LIKE-A List关系的类Set，如何用一个类的代码（用C++）表示另一个类的可重用代码？我们发现Set“很像”List：它们具有相同的包容器特性，并且它们的特性绝大部分是共同的。我们也发现Set IS-A List是不对的，等等。这两个类实现之间的相似性可能有重用机会，这

一点是很重要的。但仅从一个类应当从另一个类中导出来判断是否可重用，可能不是一个好的“鉴别器”。如果有第三种抽象用来获取这两个类的公共信息，那么这两个类可能有能力约束系统设计类型的关系。如果有 Set IS-A Collection 关系，也有 List IS-A Collection 关系，那么 Collection 可以作为这两个类的基类而使用，包括 Set 和 List 的公共代码和数据。

代码也能够用把一个类嵌入到另一个类的实例来共享，并且把类成员函数由外部转发到内部。Set 能够根据 List 这种方式来实现，无须使用派生。对象的层次结构（与类的层次结构相反）能够实现抽象重用，甚至在没有 IS-LIKE-A 关系的情况下。有时，这样的使用被称为**实现层次结构**。

这里，我们展示与重用有关的子类型和继承之间的相互作用。我们首先展示的子类型/导出类的关系从一般的可重用库观点来看是弱结构的。其次我们要从确切的重用到底是什么的观点来寻找重用：设计、代码和存储。所有这些都具有潜在的重用，但某些部件可能比其他部件具有更潜在的重用。也有些借助于它们自己的良好的面向对象设计或 C++语言，可以使那些做不到重用的部分也能够被重用。在对象范型下，设计重用是在重用上获得成功的基础。代码和存储重用也是有价值的，这两种重用能给重用概念作一些指导和启示。模板或许最适合 C++代码重用，并且通过对模板的优化而对重用作一些细化。本章集中在转发、继承和重用的关系以及封装和 ADT（抽象数据类型）上，最后我们将翻新和概括一些经验规则。

7.1 所有模拟的分解放到何处

当继承反映类与类之间的 IS-A 子类型时，软件重用就具有连带性（重要的!）好处。在类层次结构中的重用经常更多地是从设计上来考虑，而不是从重用本身来考虑。即，与应用领域子类型有关（见第 6 章）。但因为面向对象程序设计的技巧与面向对象设计结构紧密地捆绑在一起，所以代码重用也是由子类型层次结构造成的。虽然 Imaginary 数类出于设计上的考虑可以从 Complex 导出，但 Imaginary 最终重用大部分 Complex 代码只有从属性好处。在这种意义上看，面向对象设计是一种有价值的重用工具，并且在导出类中基类特性的重用是一种有价值的设计工具。

在对象范型下的重用考虑必须是在设计体系结构期间应用分析的基础。我们要求类是重用的单位，但我们也要求把这些单位精确地映射到应用领域上。特定应用的类的合适的设计与通用的重用设计之间是一对矛盾。对象范型的一个要求是系统问题的结构和解结构要同构。当我们在系统分析期间标识实体时，我们倾向于将现有的应用与此相配合。这种设计的定位，为在特定的程序或系统中的使用创建了合适的类，但对广泛的重用一般来讲却不太充分。采用问题领域/解领域同构，因为太表面化而妨碍了软件可重用。高效的重用是以整个领域的广泛设计作用域为基础的（见 6.3 节）。

作为一个例子，我们来考虑一下在设计文本编辑器中构成的一些类。其中的一个类就是 File。我们可能基于 Window、Editor、Keyboard、Session 和其他类的需要来建造 File 的特征。做这种事情的一种方式是把这些类拟人化，有些设计者通过类和动作之外的脚本起作用，或者设计者通过正确观察应用领域中的事物的相互关系的能力来领悟到这种实际需求的内含。

在编辑器构建类时，设计决策能够以几种不同的方式使重用受到潜在的损害。与 File 方式有关的一些问题是设计者能够预料到的：File 在不同的应用中可能有细微的不同。当我们对 UNIX 文件的 File 挑选语义时，另一个更通用的观点是使 File 作为带有 UnixFile 的导出类的一个抽象基类。这种通用观点为可重用 File 类打下了坚实的基础，File 类能够像 VSAMFile 那样以递增的方式扩张导

出类。

另一些问题与精确地了解 File 需要做什么有关。对编辑器来说，一个 File 需要能够读和写。高级编辑器可以在类 File 中设定检查点的能力，以支持在编辑会话期间系统崩溃时执行恢复工作。但更通用的重用 File 可能需要检查一致性，或压缩或加密，但在编辑器版本中没有一条是可以提供的。当然，编辑器版本可以包括一些在更一般的应用中所不需要的软件包，诸如检查点软件包。

下面是有些争议的说法，即考虑在所有公共可用的类型上的所有公共可用操作的联合（union）是设计中的一个缺陷。首先，要设计最好的通用文件接口是不可能的；若许多类型的内含都存储在一个文件中，并且许多合理的基本操作都在这个文件中执行，那就太复杂了。考虑到 UNIX 文件必须支持在编辑器中、在编译器（源文件和对象文件）和在操作系统中有用。第二，“好的”设计无须做许多“胖”的工作，即代码与应用无关（代码独立于应用）。

这些问题的一部分能够通过在抽象的高层上重用的表示来解决，并且实现了适合使用的上下文中的抽象。例如，无论 File 是用 ASCII，EBCDIC 构成，还是用 FILEDATA 字符定义，都能够使 File 做实际的工作，这些工作用 File 本身的行为来编码。具有这种目的的含义是参数化多态性，其中 C++ 提供了它的模板特征。这就产生了重用的层次，从而使工程师更贴近设计问题，而不是实现问题。这些折衷的接近能够把设计决策捆绑在一起，使之更灵活，能力也更强。

重用的抽象层次的另一种方式通过抽象基类产生。一个类能使用纯虚拟函数去描述可重用抽象的框架，并且该函数能够通过类用户在应用抽象的上下文处补缺。重用的抽象基类和框架将在第 11 章中加以讨论。

7.2 设计重用

我们应当在设计时而不是在实现时寻找公共的可重用的成分，以作为重用的“合理”的层次。集成重用设计的应用领域类型的分类结构，这样的类接口就应当仔细地考虑语义，因为这种语义（不是实现）被认为是可重用的成分。有关重用的经典文章有 Tracz[7] 提供了一些重用和面向对象设计之间的相互作用的看法：

软件组件的适应性（或潜在重用）依赖于领域分析执行程度和模块反映重用的参数化程度。此外，参数化格式由程序设计语言提供，可能不总是支持所期望的适应性（或重用）的格式。Ada 类属支持重用，但不能替代继承。单靠类也不能充分支持重用。当某种语言特征做了帮助开发可重用的软件时，语言本身是不能够充分解决可重用问题的。

通过对整个而不是现有应用领域的研究，类能够作为编写应用程序的主要元素。要造就好的领域分析是困难的，并且会耗费相当多的时间，系统设计只能通过一些类的领域分析来提供。细致的观察有利于在继承层次结构顶部概括一些类，特别是概括抽象类。导出类可能对单个应用是具体的。这些类是从一般中挑选出来的，可重用的抽象在储存库中应当有详细的文档和发布；如果程序员不能够寻找到类，那就不能被重用。

设计重用的含义是有以下两种情形之一：一种是已定义好的包的主要重用；另一种是类的特征的重用，即可重用语义单位与类行为的集合有关。我们再来考虑外表上可重用的 File 类。UNIX 操

7 参见 Tracz, Will. “Software Reuse Myths.” Software Engineering Notes (ACM SIGSOFT) 13, 1 (January 1988), 17-20.

作系统文件可以被创建、删除，顺序地读和顺序地写，它们也能随机地读，检查文件系统的系统性故障、重新命名、压缩等事件中的一致性。编辑器只需包含这些事件中的子集就够了。例如，编辑器无须进行文件的一致性检查，或者也不需要压缩、随机 I/O 和重命名。如果在重用命名中带有这些“包”，则必将添加程序的复杂性和加大程序的长度。把 File 层次结构分解到由用户提供的一大批类中，采用合理地选择类层次的方法，程序就能找到所需的抽象，而无须添加太多的辅助软件包。如何预测不同的应用，使用一种像 File 一样的抽象，并创建与功能相适应的抽象，这是设计合理可重用的出发点，当然，需要花费一定的时间，并且只能事后才能真正认识到。

设计的重用在系统设计阶段就能迅速地减少花费。此阶段（至少应当是）与编码的生命周期部分相比更大。只要给出所找到的重用设计，我们就能够重用已有的实现，这些对支持我们的设计工作是必需的。或者我们可以选择重新实现所有或部分最适合工程的应用的需求（可重用部分），清除应用中不必要的部分。例如，图形所支持的包，应当在不同的平台上支持不同的基本图元的硬件的实现。图形应用也应当在不改变这些平台的基础上运行。重用以不同的方式实现相同的设计：在不同的上下文中实现重用的应用。但这些重用应用比可移植性所遇到的问题要少(是重用应用而不是库的应用)。更重要的重用是，支持不同应用中富有生命力的持久性。

因为重用能够节省设计花费，所以，可以补偿在未来项目中可重用新类所花费的时间。也就是说，时间必须花在刀刃上，即花费在寻找应用领域、文档以及更通用的新的抽象上。节省时间本身就是对项目的一种长期投资。

7.3 4种代码重用机制

有时我们在优良的设计中，有时，可能由继承诱导出的子类就可获得有意义的和满意的重用结果。但就重用本身而言，典型的继承就是大家所熟悉的#include 方式——这种机制是自动地选择一个类的一部分，并合并到另一个类之中，在共享对象代码的同时也大大减轻了拷贝源代码的麻烦。

代码重用是出现在解领域中的一种现象。我们如何使用面向对象的工具而获得重用，且不需要搞乱类的集合（当然，这些类的集合较完备地集成有关应用领域的各个方面的信息）？第一个机制是转发，其中新的对象是建立在已有对象的协作基础上，而这两者合在一起就成为完成系统的一种角色。特别地，对类有准确的 IS-LIKE-A 关系但在缺少 IS-A 关系时，也是很有用的。Set 和 List 的例子（见 6.5 节）再一次说明了这一点。转发自然地与数据抽象结合在一起：Set 类能够借助于嵌入在 List 对象内的 List 对象进行运作（见 6.5 节）。Menu 重用 Window 代码是另一个例子。

设计重用的单位是类接口；类接口是缺省的重用单位。那么代码重用单位应该是什么呢？它们经常随着类的划分而发生变化，并且作为“自然”抽象的类也在继承模块中当做可重用的块而被使用，我们将立即重新访问继承。但是，转发给另一个类的对象所选择的成员函数提供了恰当的重用单位，即单个的成员函数。

代码共享是第二种机制的含义，C++不直接支持代码共享，而在 Actors，Self 是支持的，并且其他少数语言是支持委托的。在委托中，成员函数可以将一个对象透明地转发到另一个对象，并且数据成员的引用能够遵从一个对象到另一个对象的规则。在委托中，将两个或更多的个别实例协作到一起作为一个出现。如果一个对象把一个成员函数委托到另一个成员函数，则第二个对象的成员函数执行在第一个对象名字空间的上下文。委托通常用于无类的语言中。C++是作为一种技术来使

用继承的，但C++没有基于类的继承的实例模拟。如果有，那就应当是委托。

委托比继承在运行时更具灵活性。我们在C++模拟带有转发的委托：一个类的对象向另一个类的对象传送几乎差不多一样的工作。虽然委托在语言中不再具有上下文共享特性，但委托仍具有其相应的灵活性。对Menu重用Window来说，它只需要包含Window对象。许多Menu成员函数（move，refresh）能够根据Window成员函数直接实现。这种在C++简单情况下的工作还是不错的，并且是支持重用的长期存在的机制，无须违反抽象机制。但是，如果在编译时不知道Window的确切的类，那么问题就发生了，即，如果我们使用Window的导出类，根据各种情况的变化来确定它的大小和确切的性质。这时，我们通常是用嵌入一个指向重用的对象的指针（例如，用Menu指向Window）来代替实例。

对解领域重用的第三种机制是私有派生，其中导出类重用它自己实现的基类的代码，无须包括作为它自己的一部分的基类接口。在语义上，在私有继承以及在一个类作为它的成员嵌入另一个类之间时存在着细微的差别。私有派生可以作为期望重用的文档的较为方便的符号，以便用一个对象的内部状态特征来辨别另一个对象的使用。例如，

```
class Resistor {...};
class Inductor {...};
class Capacitor {...};
// 低通滤波器对它的结构而言重用了现成的
// Resistor, Inductor 和 Capacitor 抽象，
// 反映出这种私有派生。
class LowPassFilter:Resistor, Inductor, Capacitor{
public:
    LowPassFilter(Resistor r, Inductor l, Capacitor c):
        Resistor(r), Inductor(l), Capacitor(c) {}
    ...
private:
    // 私有数据不反映可重用，但刻画了该类对象的状态：
    Frequency w0;
    Quality q;
    double High3dbPoint, Low3dbPoint;
};
```

这个例子反映了私有派生，虽然这种方法有严格的限制。因为没有一个类可以作为直接的基类而出现两次：

```
class Arm {...};
class Leg {...};
// 这是非法的：
class Robot: Arm, Arm, Leg, Leg {      // 对重用的继承
private:
    Direction direction;               // 状态变量的成员
    Velocity velocity;
public:
    Robot ();
};
```

最后一种重用机制是参数化法（参数多态性），或许是影响所有这类机制的最高级的潜在重用。

下一节我们将集中讨论C++如何支持参数化法，并且如何用C++参数化法类型使程序员的习惯用法能够减少所产生的冗余代码的问题。

7.4 参数化类型或模板

C++模板特征实现了参数化类型，模板定义了类型和函数的家族。模板是对代码重用继承体系构造的一种选择。例如，假定我们要在所有类型（int，Window和其他）的栈的Stack实现中重用算法。继承方法把所有公共代码放到基类中，同时我们要创建（intStack，WindowStack等）Stack的每种类型的导出类。但是，因为该代码在Stack内操纵栈的元素，所以根据栈上的事件的类型似乎很自然地表达了这些算法。这就意味着大量的代码不能在所有Stack类型上跨越，而必须一个一个地重写。我们甚至不能有Stack抽象基类，因为函数的说明（像pop和top一样），甚至连区分栈是什么种类也不知道。

一种选择是根据无类型的void*变量来实现Stack：

```
class Stack {
private:
    struct Cell {
        Cell *next;
        void *rep;
        Cell (Cell *c, void *r)next (c), rep (r) {}
    } *rep;
public:
    void *pop() {
        void *ret = rep ->rep;
        Cell *c = rep;
        rep = rep ->next;
        delete c;
        return ret; }
    void *top () {return rep ->rep; }
    void push(void *v) {rep = new Cell(rep, v); }
    int empty () {return rep != 0;}
    Stack () {rep = 0; }
};
```

这样一种Stack的使用在句法上是不方便的，因为它仅存储了void*，从而导致了在应用代码中出现了许多“与”符号和计算。更重要的是，它丧失了大量的在类型安全性上所需要的东西，甚至更深层次的问题可以从弱类型中产生。考虑对类List添加sort成员函数用void*来实现。比较两个元素用List::sort方式是不方便的，但却是很清楚地进行了表示。

高层面向对象语言像Smalltalk和CLOS那样，用在运行时改变更多类型决策方式侥幸构成了重用。它们的多态性与C++多态性相比对继承层次结构的依赖更少，其好处是在这样的类之间存在着低度耦合，并且在运行时有义务找到“方法没有发现”这类错误。

在C++中，我们能够解决这些问题，无须牺牲类型的安全性和运行时的效率。我们在C++要做的事情是具有另一类型的参数化Stack。例7.1的Stack担当了Stack具体的种类制造的模板。该模板根据某些一般类型*T*来编写，其中*T*被假设为具有所有该对象要求的Stack特性。当我们要求有

具体类型的对象的 Stack 时，就指示编译来建立它，并且在这个模板体中对每个出现类型参数 *T* 替换此类型的特征。这就是称为从一种类型到另一种类型（带有第三个参数）的实例化[2]。int 的 Stack 能够通过用 int 作为参数的 Stack 而实例化：

```
Stack < int > anIntegerStack;
```

我们现在来说明以下情况：

```
Stack < Window > windowManagerStack;
int main() {
    Stack <int> anIntStack;
    anIntStack.push(5);
    int i = anIntStack.pop();
    Window w(0, 50, 4, 4, 4);
    windowManagerStack.push(w);
    ...
}
```

当使用模板时，要遵循以下规则：

（1）不能把一种参数化类嵌入到另一种参数化类之中；

（2）在参数化类内的静态成员是特殊的类型的实例化。

例 7.1：Stack 的模板。

```
template <class T> class Stack;
template <class T> class Cell {
friend class Stack <T> ;
private:
    Cell *next;
    T *rep;
    Cell(T *r, Cell <T> *c): rep(r), next(c) {}
};
template <class T> class Stack {
public:
    T *pop();
    T *top() {return rep ->rep;}
    void push(T *v) {rep = new Cell <T> (v, rep); }
    int empty() { return rep == 0; }
    Stack() {rep = 0;}
private:
    Cell <T> *rep;
};
template <class T> T *Stack <T> :: pop() {
    T *ret = rep ->rep;
    Cell <T> *c = rep;
    rep = rep->next;
    delete c;
    return ret;
}
```

2 该术语的使用是不太恰当的，同样的术语习惯用于从一个类中创建一个对象。在 C++中，这两种类型是通过各自的程序设计语言构造来实现的，虽然在富于哲理的层次上可以想像它们之间是有关的。

类刚好是一种能够通过参数化重用的抽象，函数也能参数化。一个通用的例子是 sort 函数，它们能对任何类型的向量进行分类（见例 7.2）。

模板在 C++中是最流行的代码重用机制，并且最主要的是它能为可重用代码构造库。但是，在 C++中，使用的是源代码重用而不是目标代码重用，这种理解是非常重要的。相比之下，基类的许多功能也能够在对象代码层上被导出类重用。对于整个代码的压缩和变化的影响，到目前为止仍有不同的看法。

例 7.2：用模板来参数化 sort。

```
template <class S>
void sort (S elements [], const int nelements) {
    int flip = 0, sz = nelements - 1;
    do {
        for (int j = 0, flip = 0; j < sz; j + +) {
            if (elements[j] < elements[j + 1]) {
                S t = elements[j + 1];
                elements[j + 1] = elements[j];
                elements[j] = t;
                flip + +;
            }
        }
    }while (flip);
}
int main() {
    Complex cvec[12];
    for (int i = 0; i < 12; i + +) cin >> cvec[i];
    sort(cvec, 12);      // 调用 sort(Complex[], const int)
    for (i = 0; i < 12; i + +) cout << cvec[i] << endl;
    return 0;
}
```

如果类模板的所有操作对类型自变量用于实例化类都不受影响，那么共享的代码就能够根据 void*来计算，并且作为能够处理必须计算的其他类型的基础。从我们要进行实例化的几个列表的类中考虑一般的 List 类模板。如果该 List 模板有一个 sort 成员函数，那么它就可能对 List 的每个实例分别建立 sort 代码，无论它是否被调用。假定我们不去分类列表，也并不立即携带 sort 多个拷贝的额外包，我们就能执行以下工作：

```
class WindowList:private List < void* > {
public:
    Window *get () { return List < void* > ::get(); }
    void add(Window *p) {List < void* > ::add(p); }
    // 未排序 … 留给私有
};
class AddressList;private List< void* > {
public:
    Address *get() {return (Address* )List < void* > ::get(); }
    void add(Address *p) {List < void* > ::add(p);}
    // 未排序…留给私有
};
```

该实现是令人乏味的，但至少对每个新的抽象不要求再重做整个的 get 和 add，并且它限制了 List<T>::sort 对一种抽象的拷贝。

如果大多数模板的成员函数过分依赖于类型参数（作为 sort 的话，必须在列表中比较两个项的大小），那么这种技术不会被接受。参数化源代码层引起了有趣的参数化问题的一个地方是具有其他类库的库类参数化。例如，假定类 Set 是根据类 List 实现的（见例 7.3）。从 Set 实例化一个新的类型要求现有的 Set 和 List 两方面的源代码。如果 List 依次依赖于另一种参数化类型，那么它就需要现有的源代码，如此等等。

Set 对 List 的使用应当对用户是透明的。需要处理 List 源代码（在 Set 公有接口中类不出现）与直觉是不一致的。这就可能导致软件管理出现意外，并且要求特别关注用户的文档，这似乎违背了信息隐藏的原则。

另一个问题是对每个被实例化的 Set 要很完美地复制 List 代码。List 实例化之间的差别可能很小，而某些未被 Set 用过的 List 代码就无论采用什么方式都能产生了。这可能使软件包复杂化：它意味着若没有 List 源代码，Set 的源代码就不能够处理。在这种特殊的情况下，一种类型可能没有得到其他方法的辅助就不能被处理。但是，通常一种类型使用另一种类型，对其实现来说不应当因重写的原因而使用户赢得了两个源代码包。

为了提高参数化能力，并且无须从 List 多份拷贝中花费复制代码的开销，我们就需使用间接模板习惯用法。

例 7.3：参数化 Set 实现在 List 项中。

```
template < class ElementType > Set {
public:
   void add(const ElementType& e) {
     if (!rep.element(e)) rep.add(e);
   }
   ElementType get () {
      return rep.get;
   }
   void remove(const ElementType&);
   int exists(const ElementType& e) const {
      return rep.element(e);
   }
   Set < ElementType > Union(const Set < ElementType >&) const;
   Set < ElementType > Intersection(const Set < ElementType >&)
      const;
   int size() const {return rep.size(); }
   void sort(){rep.sort(); }
private:
   List <ElementType> rep;
};
```

例 7.4 展示了 List.h 文件定义一个简单参数化 List，习惯用于实现新的 Set 类。ListItem 类是一个辅助类，它包括的项（item）已经插入到列表中，并且还包括链接列表结构本身的一个指针。List 维护了这些项的单链接列表，以作为它的链表结构的内部表示。

在独立的文件 Set.h 中，我们说明模板 Set，对 Set 实现使用了 List。我们要避免对每个参数化

Set 在两个地方用添加信息隐藏层次的方法实例化 List 类。首先，每个特别的 Set 种类由类 SetBase 导出。这个 SetBase 接口含有要求展示其元素的 Set 对象的所有操作：

```
class SetBase {
friend ErsatzListElement;
private:
   virtual int comparelt(const void*, const void* )
      const = 0;
   virtual int compareeq(const void*, const void* )
      const = 0;
};
```

这些操作说明为 virtual，并且要从包含在集合中的对象类型中消除相互影响：它们与外部世界的通信仅通过 void*类型进行。最后，这些操作将从 Set 模板中定义“真实”的类实例化，根据包含 Set 的类型对象来定义。

例 7.4：List.h 文件。

```
template <class T> class List;
template <class T> class ListItem {
friend class List<T>;
private:
   ListItem <T>  *next;
   const T item;
   ListItem(const ListItem<T> *n, const T &i);
      item(i),next((ListItem <T>* )n){}
};
template <class T> class List {
public:
   void sort ();    // 排序链表就位
   void put (const T& t){head = new ListItem <T> (head,t);}
   T get() {        // 从链表头中获得
      T retval = head ->item;
      ListItem<T> *temp = head;
      head = head ->next;
      delete temp;
      return retval;
   }
   int element (const T&) const;  // 成员关系检测
   ...                            // 许多其他有趣的操作
   int size()const;               // 元素计数
   List():head(0) {}              // 缺省构造函数
private:
   ListItem<T> *head;             // 链表头
};
```

第二种技巧是在另一个类中隐藏列表元素 ErsatzListElement，只通过 void*指针来访问：

```
class ErsatzListElement {
   // 这是未参数化的，其中的意思是有一个对
   // 所有集合服务的列表模板实例化
public:
```

```
    void *rep;
    int operator < (const ErsatzListElement& l)const {
        return theSet ->compareIt(this,&l);
    }
    int operator == (const ErsatzListElement& l) const {
        return theSet->compareeq(this, &l);
    }
    ErsatzListElement(const void *v = 0):rep(v){}
    ErsatzListElement(const SetBase *s, void *v = 0):
        theSet(s), rep(v) {}
prirate:
    const SetBase *theSet;    // 恰好用于访问比较两个对象的虚拟函数等
};
```

这种包装类（wrapper class）的实例由 List 习惯用于实例 Set 的方法来维护。这就意味着所有 List 将用同样类型的 List<ErsatzListElement>的集合来创建，所以，一种列表类可以通过跨越所有 Set 来共享。

例 7.5：灵活、高效的 Set 模板。

```
template <class T> class Set:private SetBase {
    // 重用的私有继承
public:
    void add (T  t2) {
        if (!exists(t2) {
            ErsatzListElement t(this,new T(t2)); rep.put(t);
        }
    }
    T get () {
        ErsatzListElement l = rep.get(); return *((T*) (l.rep));
    }
    void remove(const T&);
    int exists(const T& e) {
        ErsatzListElement t(this,(T*)&e);
        return rep.element(t);
    }
    Set <T> Union(const Set <T > &) const;
    Set <T> Intersection(const Set <T > &) const;
    int size() const {return rep.size(); }
    void sort() {rep.Sort(); }
    Set();
private:
    List <ErsatzListElement> rep;
    int compareIt (const void *v1, const void *v2) const {
        const T *t1 = (const T*) v1;
        const T *t2 = (const T*) v2;
        return *t1 < *t2;
    }
    int compareeq(const void *v1, const void *v2) const {
        const T *t1 = (const T*) v1;
        const T *t2 = (const T*) v2;
```

```
        return *t1 == *t2;
    }
};
template < class T > Set < T > :: Set() {}
```

最后，就有了 Set 模板本身（见例 7.5）。它是带有包含在对象类型的参数化，但除了 List<ErsatzListElement>外并未创建新的类型。它包含了比较类 *T* 的两个对象的私有成员函数，并且无论什么样的操作 Set 都要求定义在 *T* 上。只有在把 void*参数转换到带有分类整理的类型 *T* 之后，这些“脚手架”函数才向 *T* 转送它们的要求。这种分类整理是安全的，因为在这种参数化类型内，能够确保在这种代码中出现的 void* 总是指向类型 *T* 的对象。当然，所有代码（“脚手架”的以及所有的）在编译中用参数化类型支持的机制自动地在模板中产生。

这是最后的 Set.h。这是使用参数化 Set 的简单应用，并逐个地使用了参数化 List。即使有两种 Set，也仅有 List 一种必须同时支持两种的实例化。

```
#include <iostream.h>
int main() {
    Set <int> foo;
    Set <double> bar;
    foo.add(1);
    foo.add(2);
    cout << foo.get() << "\n";
    cout << foo.get() << "\n";
    bar.add(3.0);
    bar.add(4.0);
    cout << bar.get() << "\n";
    cout << bar.get() << "\n";
}
```

现在，Set 抽象能够作为独立实体被管理和使用。List<ErsatzListElement>和 List 抽象的对象代码能够提供给 Set 的源代码。

当使用这种习惯用法时：当模板大范围地用于程序中时，要使用这种习惯用法。特别地，当一个参数化类使用了指定另一个参数化类的实例的类型参数之一时。

不同的 C++的安装可能对管理过程的集合的要求稍微有所不同，以确保 List 实例只发生一次。一种典型的实现方法是在它们自己的文件（具有.c 后缀）中放入 List 成员函数定义。

7.5 私有继承：继承支持重用吗

在 6.5 节中，我们看到了由 CircularList 导出的 Queue 继承的不恰当的应用。Queue 最后在它的特征中有一些意想不到的操作，因为它是从其基类继承过来的。这是由公有派生所造成的。为了解决这个问题，我们承认 Queue 重用了 CircularList 功能，无须与另一个类交换使用。重用不一定违反 CircularList 抽象，也不必紊乱 Queue 的抽象。我们至少有以下 3 种方法可以实现：

（1）把 CircularList 放入每个 Queue 内（实现层次结构）。

（2）对匹配的 CircularList 对象使用委托模型，并具有 Queue 委托的一些操作。

（3）使用私有继承。

第一种解法是由 Liskov 提出的：它不违反数据抽象，并且它是重用的直觉模型。该方法被称为实现层次结构，认为实现层次结构就是重用类的实例的层次结构，但这要与类继承的层次结构的“实现”区分开来。实现层次结构是一个类的对象在另一个类的对象内部的包容器。重用所使用的实现层次结构不作为继承的子类型联系而使用。这就表明，若我们能够重用 CircularList 的功能就不要求能与另一个类交换使用。下面是一个使用实现层次结构的例子：

```
class CircularList {
public:
   int empty();
   CircularList();
   void push(int);    // 压入头部
   int pop();
   void enter(int);   // 压入尾部
private:
   cell *rear;
};

class Queue {
public:
   Queue() {}
   void enterq(int x) { l.enter(x); }
   int leaveq() { return l.pop(); }
   long empty() { return l.empty(); }
private:
   CircularList l;
};
```

这是一种转发的形式，需要强调的是，（1）Queue HAS-A 列表（List）；（2）不应用 IS-A 关系；（3）所讨论的 CircularList 不被共享。这里，没有违反 CircularList 的封装性。

第二种方法——委托——是可行的，但在 C++中却很勉强，缺少主语言的支持。

第三种方法——私有继承——可能使用以下方式：

```
class Queue:private CircularList {
public:
        Queue() {}
   void enterq(int x)   {enter(x); }
   int  leaveq()        {return pop();}
        CircularList::empty;
};
```

私有继承和实现层次结构或多或少地与效率相关，每种方法都有它自己的优点。在导出类的情况下，能够直接地使用基类的函数，程序员必须提供在基类那里处理的操作的转发。

导出类能够重写在私有基类中被称为虚拟函数的功能，所以，私有继承与两个类协作的重用包容器相比执行得更好。基类定义了整体结构，并且特性受作为纯虚拟函数的导出类的约束，因为导出类能够引用（和重用）基类成员函数。一个较好的例子（见第 11 章）是 TaxForm 框架。

但重用的派生也有一定的缺陷。客户类能够访问私有基类的 protected 成员，这就违背了封装性：封装是为了保护内部成员。而继承又有容易使人联想到子类型联系的潜在危险。

包含私有继承和转发的混合方法也可以被使用：

```
class Queue:private CircularList {
public:
   Queue() {}
   void enterq(int x)     { enter(x);}
   int leaveq()           { return pop();}
   int empty()            { return CircularList::empty();}
};
```

也许可以找到在风格上比其他选择更巧妙的方法，对 empty 的引用产生了性能的额外函数调用的补偿（除非 Queue::empty 的主体被内联扩充）。如果出现了重载函数，那么这种构造必须采用访问标识符技术来代替。同时，程序员希望在同名不同基类的成员函数上采用不同的访问方法。

当私有继承特意用重用替代子类型时，由于大多数 C++编译出带有标记的公共错误而获得了诊断消息。从体系结构的观点来看，这是很重要的。我们再来考虑一下 CircularList 的例子，除非具有虚拟函数：

```
class CircularList {
public:
   virtual int empty();
               CircularList();
   virtual void push(int);    // 压入头部
   virtual int pop();
   virtual void enter(int);   // 压入尾部
private:
   cell *rear;
};
class Queue:private CircularList {
public:
      Queue() {}
   void enterq(int x)     {enter(x);}
   int leaveq()           {return pop();}
      CircularList::empty;
};
int main() {
   CircularList  *alist = new Queue;       // 编译时出错
   return 0;
}
```

如果执行是合法的，那么 Queue 能合法地出现在 CircularList 希望的任何上下文中。但是，编译器标识出了对象实例化的一个错误，从而获悉 Queue 不能请求执行 CircularList 的工作。

在类对象指针或引用的上下文中，只有当使用虚拟函数时，这种问题才得以进一步考虑。

7.6 存储重用

有时，我们需要共享几个其他类或对象的现有类的行为，而无须承受超过一个共享类拷贝的存储开销。例如，几个 Window 对象可能希望对几个与输入有关的函数共享单一的 Keyboard 对象。在

每个 Window 中用嵌入到内部的 Keyboard 代替，所有 Window 能够进一步用单个的公共的 Keyboard 类的操作来代替。当重用公共存储时，就可用 Keyboard 的行为赋予每个 Window。处理/主体习惯用法（见第 5 章）能够反映共享的使用：主体可以被共享，而每个处理能够代表的对象如“使节（ambassador）”或“代理（surrogate）”。有时，static 成员是在所有给定类的对象之间共享存储的最好方式。

我们也可以决定更基本的体系结构以考虑其他的共享存储问题。例如，把一个对象放入到一个比 CPU 还要大的公用内存之中。这在 C++中能够直接执行，提供的对象类没有指针。类中的显式指针变量应当既可以避免也可以专门地得到处理。指针也可以出现在其他外部中：引用成员和虚拟函数的许多实现也使用指针。指针的内容必须能够在每个应用的上下文中赋予正确的解释，通常的情况是，只有当它宣布为另外一个共享内存对象时才这么做。如果每个处理器（或过程）在它的地址空间中对相同的逻辑地址映像出共享内存段，那么共享指针就会产生很少的问题。但大多数操作系统建议不要依赖于这样的映像。

若使共享信息没有指针，并且把所有指针和虚拟函数语义放到信封类中，就能使共享内存在 C++应用中被使用。仔细地设计和调整与使用指针有关的类，并且在此类中无虚拟函数，也能够在共享内存段中使用。

7.7 接口重用：变体

一个柔性设计系统能够把同一设计的几个不同的实现合并在一起，每种实现可以认为是相同类型的实现，虽然内部的算法有所不同。这样的例子包括：

- 远程过程调用：“代理”类成员函数可以转发远程类的所有网络要求。远程和本地类都有相同的特征，但其实现几乎没有共同之处：本地类只把事务送往远程类。在高层上，该操作的语义是相同的。

在网络应用中，可能存在 3 种不同的给定类的变化：

（1）实现工作本地化，即，本地服务器类。考虑一个文件服务器对象：如果该对象知道这个磁盘与本地机器相连接，那么就直接地执行该工作。

（2）向远程机器转发要求来实现变化。这里，本地文件服务器对象知道该磁盘是在一台机器上，并把网络媒体的要求转发到另一台机器上，这台机器就用远程服务器类来处理。

（3）远程服务器类。这个类逻辑上类似于本地服务器类（它可以嵌入本地服务器对象自身之中），除非它从网络上接受其要求。它不可能考虑其他两个变体的真实性，因为它的直接接口是网络的协议，而不是针对用户的。

- 市场依赖性：在不同的上下文中，一个类可能需要采用不同的实现。例如，可以存在一种 Console 类，但我们要求说汉语的市场与说英语的市场具有不同的实现。
- 硬件依赖性：例如，Stack 类可以在一种机器上直接利用栈硬件来编写，并且它也可以在另一个具有可移植方式的环境中编写。如果该类由库来提供，那么该类接口定义对两种实现就是相同的，并且这两种实现是等价的。这同样可以应用于依赖于硬件的原语（诸如浮点操作）。
- 测试设备：类的一个版本可能是一个测试版本，另一个版本是调节高性能版本。

- 性能调节：不是引用计数的现有的具体数据类型可能对系统的一部分，要求引用计数副本，其中，内存空间和实时是关键所在。相关的例子见3.5节，此例子是通过使用简单数据抽象来实现的，但使用继承要重新实现。

这些例子的共同之处在于变化仅仅弱对应于应用中的实体；它们更贴切地映像为实现。但是，继承是实现这些例子的便利的工具，特别地，如果使用抽象基类的话。每个导出类是作为抽象数据类型的特殊实现，表示了共享抽象基类的特性。我们重用单个设计，并在几个不同的实现中反映了这个设计。

这样的导出类称为**变体**。导出类的名字是非本质的，并且对这样名字的引用只限于局部。对这些类的对象的访问通过指针或引用，使用虚拟函数。客户代码根据抽象基类来编写。

另外的选择是，在父类内处理可替换的情况，而不是从父类中导出新类。这就是封装重用的结构，并且当在这种上下文使用的情况下被称为样本（examplar）。在要求平台给定操作时，样本类可以转发对它自己对象中的一个要求，样本类本身使用虚拟函数处理此要求。样本派生的优点是封装变体本身实现的名字。例如，我们提到的 Stack 类可以用两种方式之一来实例化：一种方式包括使用专门的处理器硬件的对象实现栈；另一种方式是使用方便的数据结构技术。样本可以认为是以同一方式来作为委托多态类的实现的（见5.5节）。即，内部对象不可以物理地包含在它所包含的对象字节之中，但对具有专门变体语义的对象来说，逻辑包容器可以从外部对象指针中模型化。

7.8 重用、继承和转发

设计者和编码者必须在重用的完美性和重用的有效性之间进行折衷，设计独立的类的封装。设计者在比 C++代码（例如，白板体系结构）更加灵活的介质中工作对重用设计和清晰地设计平衡的分配功能和数据特权更为有利。一旦这些设计减少了代码，语言就强迫在类的自包容器之间建立一个扩充，并使另一个类能够具有重用它的代码能力。

如果重用实现使用私有派生，或者用嵌入实例方法（即该实例嵌入到另一个对象的每个实例中）实现的话，那么两个类之间的耦合就能够确定。当采用派生时，基类的封装能够使用 protected 成员部分地对导出类开放。若试图在不使用私有继承的情况下共享符号，就要用 friend，即可能放弃一个类对另一个类的大量的可视化选择。共享符号对 friend 和继承都没有好处，它会导致类公有接口的过度增长。Snyder[3] 认为：

这种解法（内联函数用于直接而有效地访问基类变量）的更为重要的目的是要求定义实例变量的父类的设计者（以及所有涉及父辈的设计者）之间进行协作，因为对被继承的实例变量不访问是可能的，除非提供了合适的操作。我们认为这是必要的要求。如果类的设计者需要访问被继承的实例变量和未加定义的合适的操作，那么正确的做法是要与父类的设计者交涉（或协商），以便能够提供此种操作。

这里，关键的重用不就是把一个类的代码给另一个类来使用吗？但作为一个类的可重用能力是要将感兴趣的部分通知其他类方可实施。“供应商”和“用户”之间的潜在的可重用代码的协商是在于引用所有的感兴趣部分，以便保存优良的设计结构，而不是单纯地要求把代码和复制减少到越少

3 Snyder,Alan. “Encapsulation and Inheritance in OOPL.” SIGPLAN Notices 21，11（November,1986）.

越好。

7.9 源代码重用结构的选择

我们有许多方法管理源代码的重用，或者用给定的应用有条件地调节某些已有的代码。继承是一种工具。对于继承和传统技术来说，都需要一定的时间和空间。我们认为在这些技术之间做一些折衷是很有必要的。

最简单的机制是运行时选择，如 if 语句和其他运行时决策性标志，用这种标志来决定两个分支代码中哪一个分支在执行时更合适。在这种情况下，我们考虑以 C 语句作为管理的单位比较合适。典型的例子是，在运行时根据全局标志的内容选择是激活调试还是诊断代码。

另一个选择是条件编辑，熟悉 C 的程序员采用#ifdef 或其他方法。这种方法对查出故障有用。这一小段代码只有在编辑时满足某些条件之后才能决定是否执行这段代码。当目标环境的选择要求有许多不同的语义和语法时，才是最有价值的。例如，I/O 寄存器是根据目标处理器的种类在不同的字长之间进行选择。在这种情况下，我们考虑把功能作为管理的单位比较合适，即便为了管理和安装上的方便而把几个功能包装在源代码文件中也无妨。诊断代码也可以这样包装，虽然失去了一些灵活性，但还是比如果运行时才能使用这样条件下的编辑少了一些开销。

极端（但也是公用）的方法是使用各自的源代码。这种方法从一开始就使用已有类的源代码的拷贝，可以利用这份拷贝来做你想做的事，之后代码的这两份拷贝就各自存在了。如果这两个类同时存在于同一个程序，那么其中一个类必须用适当的名字重新命名，以便保持类的区分。即使这两个类加以改变但仍然有许多共同之处：类接口可能没有什么大的改变，并且内部数据结构甚至于整个成员函数在改进中可能未涉及到。但这个方法的重要缺陷是，如果原始作者把固定的错误应用于原来的源代码，那么就不可能发现这些错误。即使接到这样改变的通知，也必须重新编辑原来的源代码；这是费力不讨好的事情，特别是如果原始作者在成员函数中做了更新，但并未触及成员函数，这种过程也是容易出错的。另一个缺陷是导致了不必要的代码增长。如果某人在同一系统内用这种方式“重用”你的代码，那么就会导致许多代码的激增。

如上所述，继承是管理类的强有力的工具。如果需要创建一个新类，这个新类在功能上很像已有的类，但实现上有点不同。继承允许新类建立在已有类的基础上，并且原封不动地共享类的其余部分。

作为重用方法的继承有它的责任。假定存在着用对象的软件产品作为构造块，并假定我们使用派生来很好地调节基于某些软件的行为，则它能适合于在不同的硬件和操作系统平台上的应用。这个产品需要在新的平台上运行，要求尽可能小地对代码进行修改。

例 7.6：平台改变的源代码特征。

```
int SesInterface::saveTty(SGTTY *tty) {
#ifndef USG
#       if sun
                return ioctl(2, TIOCGETP, tty):
#       else
                return gtty(2, tty);
#       endif
```

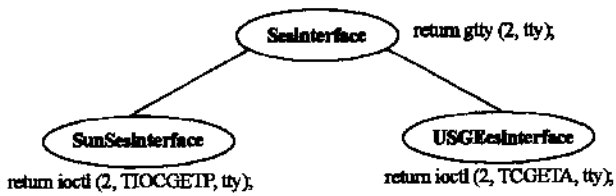
```
#else
        return ioctl(2, TCGETA, tty);
#endif
}
```

例如，考虑例 7.6 的对话接口的例子。SesInterface 是在某些交互应用中表示键盘和窗口功能的类的一个例子。SesInterface 类有一个成员函数，SesInterface::saveTty(SGTTY*)为该接口设置驱动特征。这个成员函数通过调用基本平台中的基本原语来工作。这些原语随着操作系统和平台的功能的不同而不同；这里，使用源代码特征将其变化加以分解。对于“USG”系统，是用带有 TCGETA 参数的 ioctl 来调用的；对于非 USG 系统，Sun 是一种特殊情况，而所有其他系统使用 gtty。

另一种选择是使用继承结构（见例 7.7），添加新的代码就意味着把新类添加到继承树的合适位置；新类继承它的父类的所有属性，保留一种变化的成员函数。使用这种方法，不存在源代码特征。一种猜想的优点可能是在改变之间的挑选能够在连接-编辑时（用选择右边对象代码文件）而不是在编译时进行。但是，这存在着陷阱。在系统中某处的代码形式为

```
void User::code()  {
   SesInterface *si = new  SesInterface(keyboard, window);
   ...
}
```

例 7.7：特征选择的继承。



不知情的设计者可能没有意识到在添加的导出类中，所有包含对父类实例化引用的代码也必须以描绘实现的变化为特征。用没有实例化的导出类来替代父类，这种改变还从来未见过。我们可以通过 User 本身及其对它的改变中来获取以上代码。

```
void User::code()  {
#ifndef USG
#     if sun
         si = new  SunSesInterface(keyboard, window);
#     else
         si = new  SesInterface(keyboard, window);
#     endif
#else
         si = new  USGSesInterface(keyboard, window);
#endif
   ...
}
```

当然，我们也能设法获取源代码的特征，对于有些情况来说，应该避免使用导出类。通过在 User 中创建 3 个新类，我们能够从这种困境解脱出来：

```
class USGUser:public User {
public:
    void code() {
        si = new USGSesInterface(keyboard, window);
        ...
    }
};
class NonUSGUser:public USGUser {
public:
    void code() {
        si = new SesInterface(keyboard, window);
        ...
    }
};
class SunUser:public NonUSGUser {
public:
    void code() {
        si = new SunSesInterface(keyboard, window);
        ...
    }
};
```

但我们现在必须回顾一下创建 User 对象的所有代码，并对包含这些代码的类做相同的事情，那就意味着这种变化可能是围绕系统的多米诺骨牌。

在这样的大多数情况中，最好继续使用源代码特征。一种选择是对类添加抽象层次，特别是在类对象初始化领域中需要一直支持易于演变的抽象层次。这种高级习惯用法在第 4 章中已经讨论过了，此处不再赘述。

7.10 在重用和对象上的概括

下面列出的 6 条都是本章已讨论过的，它可以作为在 C++代码重用机制中如何考虑使用继承的比较好的对照核对表：

（1）领域分析是类可重用的重要因素。应用领域应当对它的参数化“压力点”进行分析，以便最大限度地拓宽可应用的范围和最小的发生剧变。类绝不是对所有人和所有事都满足的实体，如果是这样的话，则对理解和使用均会产生巨大障碍，就更谈不上重用了。无论怎样，类应当满足当前的应用。

（2）用类接口来定义的设计重用，或许对提高大型项目的生产率有极大的潜在的好处。接口重用打算抛弃掉或者在逐个实例基础上重写部分实现，所以不可能存在许多代码的重用。但是，对重用有意义的代码仍然有可能存在，并且允许公共部分进行演变。

（3）在支持重用的对象范型的能力上，关键在于抽象基类。类层次结构相对于问题空间中的较大的不相交类型的联合，提供了高层抽象，概括了广泛而不同的应用。使抽象类获得可重用的关键在于潜在地拓宽它的领域范围，由 ADT（抽象数据类型）的单一实现提供。但这并不是说仅由抽象基类来单独解决重用问题，合适的领域分析及洞察应用的需求仍然是很重要的。

（4）当实现重用的惟一目的是使用继承时，就应当使用私有继承。公有继承反映了子类型的联

系，但不是重用。

（5）如果设计重用，就要在许多小的类或根据应用领域的建议所构造的少量的大型的类之间进行折衷。在现有应用中所推荐的较大型的抽象类，可能在这类应用上容易使用，但在相同领域中可能对其他的应用未必能很好地使用。与上下文无关设计的小的类，增加了可重性的范围。在传统的Smalltalk类层次结构中，重用可以达到最好的境界。但是，类的最好的重用粒度是否可用，只有在程序设计环境存在强大的浏览机制时才可能确定。同样地，最大（粒度）的可重用类可能并不是应用中最感兴趣的那部分；可重用类更多地充当“支持施动者”的角色。

（6）用模拟运行时类型来支持类似于委托的模型，该模型消除了许多在编译时类型限制重用的约束。支持这种模型的习惯用法在第6章和第8章中加以描述。

练　习

7.1 讨论以下断言：“重用是思维的替代物”。

7.2 重新作为变体来实现3.5节的Stack例子。

7.3 列出应该在每个C++系统库中获得应用的可重用类。模板占这些类的百分之几？

7.4 注意，在习题7.3中的类之间的依赖性（依赖性包含了由一个类参数化另一个类，一个类派生出另一个类，或者一个类的代码被其他任何一个类使用）。依赖性结构的深度是怎样的？是好（就重用而言）还是坏（就应用程序员开销而言）？

7.5 通常每个C++系统应该提供的可重用函数是什么（前两道习题的类库是独立的）？这些函数中有多少个是模板？在类库中的这些函数有多少个使用其他函数？

7.6 编写类模板 CountedClassObject<T>，创建一个非引用计数类 *T* 的计数类对象的抽象（提示：用3.5节中的CountedStack作为模型）。

第8章

C++中的样本程序设计

大多数C++类型系统都依赖于编译时的构造。静态类型检查可以尽早地发现接口错误，并能产生效率更高的代码，这比推迟到运行时检查或根本不做检查的代码要好得多。但是，支持运行时类型检测需要借助于面向对象程序的能力及其灵活性。使用虚拟函数的程序可推迟运行时类型的部分检查工作，并允许多类的对象可以互换，但这种开销直至运行时才能确定。如果我们在 Telephone 继承层次结构中对类的对象采用 ring 操作，就不能确切地知道将要发生的情况，而仅知道某些合理的操作在运行时发生。

其他的面向对象程序设计语言，诸如 Smalltalk 和嵌入到 Lisp 的面向对象语言家族，在我们所感兴趣的运行时灵活性中，对编译时的类型检测作了进一步的让步。通常这种结果会产生效率不高的代码，并且如果一个对象执行了它所不了解的操作，那运行时的意外就可能出现，然而这样却有益于灵活性和多态性。强面向对象 C++程序一般需要比只由虚拟函数提供的程序有更高级的运行时类型检测。

例如，假定我们要从磁盘文件中读 Number 对象。该对象可以作为 Complex，BigInteger，Imaginary 或某些从 Number 导出的其他类而向磁盘写入，但程序员在处理 Number 类层次上的数值时并不知道也不可能注意到。我们要从磁盘映像中“重新构造”Number 的实例，这个对象必须在构造它本身时把类型特性（Complex，Imaginary 等）放在首位。

C++的习惯用法可以应用这样的灵活性。样本（exemplar）[1]习惯用法就是本章所涉及到的内容的一个例证。样本习惯用法也使得对象在执行演变其类型特性时成为可能。例如，当将 Number 值从复数(5-3i)改变到 2 时，就可以建议它的类型从 Complex 改变为 Integer。如果要保留 Complex 的身份，就要继续使用复数算法以便获得更简单的结果。

本章描述有多少个编译时的类能够在运行时用特定的对象（称为样本）来接替，以提供运行时超越 C++的直接支持的类型的灵活性。为了更好地理解样本的习惯用法，我们应稍微回顾一下程序设计语言的背景。

松散地给面向对象程序设计语言分类的一种方法是根据它们的类是否作为基本的和独特的构造来区分。C++是这样一种语言：类在对象是什么和对象如何工作的语言模型中扮演着基本角色；绝不能在此类范围之外来讨论对象。在对象模型中，类作为中心作用的语言我们称为**经典语言**。

在 C++中，类和对象是两个不同的概念：类的行为像类型（它们都实现抽象数据类型），而对

---

1 这个概念通常称为原型（prototype）。这个术语隐含着语言支持的 C++缺少的委托。原型也包含了通常的“（制造）模型”或“抛弃”的含义。为了避免因使用原型而带来的混淆和不精确，本书用术语样本来代替。

象是使用声明或 new，在这些类型和内部类型中创建实例。类在编译时是不变的，而对象只有在编译时才存在。用户可以使用类或内部类型去创建新的对象，但不能只从一个对象来创建新的对象。在 C++中对系统的编写有类的聚合，通常认为是作为在层次结构（根据基类和导出类的组织）中的布局，并且类有各自的对象聚合，也可以认为是在层次结构中的组织（一个对象的存在导致了另一个对象的存在）。类是截然不同的。例如，类不是一个函数，也不是一个对象，更不是一条语句。同时具有类和对象的不同概念的语言称为双重分级结构语言，而 C++正是属于这种范畴。

某些语言，诸如 Self，是单一的分级结构语言。替代类的是对象的“因子”，某些对象按照惯例是作为样本来设计的，即从其他对象拷贝而来，或者“克隆（clone）”而来。克隆之后，这些对象可以修改并可以用改变原始类的属性的方法（如成员函数的行为）以及与对象实例的关联来加以调节（改变实例变量的值）。样本对象既可以是单一的分级结构语言，也可以是双重分级结构语言。样本对象的这种灵活性是有用的，对于程序的演变、对于重复开发、对于连续运行系统的更新以及对于运行时类型支持程序设计风格都是重要的（例如，人工智能的应用）。

单一分级结构语言支持创建初始样本对象机制，即为了创建一个初始化默认或空对象，并对此对象添加域和与此对象相关联的成员函数的机制。C++没有这样的机制，或至少没有提供在运行时递增式执行这种方式。所以该样本是对编译时不变的类的引导。一个类可以创建多个样本对象，每个样本对象的初始化是用专门数据来定制它的类型特征。例如，单一类的语法分析遍数，既可以有 8 遍分析也可以有 10 遍分析使其实例化。它可以用任何实例作为样本，并且任何样本可以作为实例，这些都是可能的。虽然可以按照通常的习惯用法指定单一样本来为某些类似结构对象的集合“类型”的角色服务。

对那些熟悉数据库概念的读者来说，程序设计语言的构造和数据库概念非常相似。我们可以把类作为数据库模式（schema）的成分来考虑：定义特性、关系、元组布局等等。模式告诉我们如何把新创建的元组放到数据库中。模式与类很相似，而元组与对象很相似。我们可以灵活地将新的元组添加到系统中，我们可以按照需求很容易地从某些类中创建新的对象。我们也可以更新元组的内容或改变对象的内容，而不会影响到模式。这样就具有了高度的灵活性。

但是，如果模式本身非要改变又怎么办呢？在数据库领域中，这种情况称为模式演变，并且不同的系统支持不同程度的灵活性。如果我们需要经常地改变模式，就要求对元组的修改在模式演变中有相同（种类）的灵活性，即，我们应当能够添加的一个关系域与我们对数据库增加的一个元组具有相同的简易性。在面向对象程序设计领域中，我们可以要求在处理类分级结构中的改变有相同（种类）的灵活性，以便在运行时能够很好地操纵对象。

这样的灵活性是对象范型中的一个重要基石，正如上面列举过的 Number 类一样，对大型系统开发要达到这样的灵活性还有很长一段路要走。例如，假定程序需要在其整个生命周期中一点一点地构建类的行为，系统就不能停下来，不能重新编译，也不能重新启动调节超出功能之外的变化，这一点非常重要。在这个类中包括大型的、嵌入式的诸如电信系统和金融市场数据库的系统。纪经业公司可以在它的软件中使用面向对象设计，其中每个账户类型用类来表示：不要求贸易社团一定要停止整个交易以改变已存账户类型的行为（例如，在给定时间内决定限制多少笔股票可以卖出的政策功能）。

大多数的这种灵活性来自于编译与运行时的折衷。在 C++中，类语义的描述与编译时有关：任何类特性的改变会导致一次或多次重新编译。另一方面，对象有运行时灵活性。类在编译时的特性

（事实上它与对象有某些不同），程序演变会造成失败。解决此问题的一种方式是把类赋予对象的特性，即类正好可能是另一种对象，它是跨越所有其“子孙”的一种表示，样本则填补了这种需要。

本章描述在C++实现中近似样本特性的习惯用法。这里，在单层分级结构之后风格被模型化了。其中，对象服务于用类扮演的传统角色；类轮流地用元类（metaclass）概念（如Smalltalk的）的角色来履行。

8.1 雇员样本的例子

作为一个例子，我们来构造一个商业组织的成员活动的模型系统。在这个系统中，典型的类包括Employee，Manager，VicePresident等。因为这些角色的定义可以在整个生命周期中发生变化，所以我们要求选择一种样本方法。

对于大型项目的程序，是从单一的公共的基类中派生出所有的类，这样的程序设计实践是有用的。例如，通用的基类可以用于诊断及其性能监控工作。如果有一个单一的公共基类就会更好地利用样本的灵活性，所以我们需要建立这样一个公共基类（称为Class），然后从Class再派生出系统中所有的类。例8.1展示出从Class派生出来的Employee类的说明。这种说明很像在C++中编写的原始类，但有个例外是，有一个附加的成员函数make，其中参数列表类似于这些构造函数。注意，这些构造函数本身已经移出该类公有接口之外（这里，构造函数对使用Employee的导出类已经移至protected部分之内）。在样本习惯用法下，当我们使用这种类时，这些make函数取代了构造函数。与“规范”的C++的类使用的其他差别是无偿的构造函数采用Exemplar对象作为参数，其解释如下。

例8.1：样本习惯用法的Employee类。

```
class Employee:public Class {
public:
    Employee(Exemplar);
    Employee *make();
    Employee *make(const char *name, EmployeeId  id);
    long printPaycheck();
    void logTimeWorked(Hours);
protected:
    Employee();
    Employee(const char *name, EmployeeId id);
private:
    Dollars salary;
    Days vacationAllotted, vacationUsed;
    String name;
    EmployeeId id;
};
extern Employee *employee;
```

于是，我们就有了此类。但正如上面所述，此类的主要目的是“类型的类型”，并且它的实例是作为制造新实例的“工厂”。所以下一步我们需要创建实例本身。样本可以静态地作为对象来定义，它具有程序生命周期范围和文件的静态作用域，通过全局指针类型所指向的在继承层次结构中高层的类来访问：

```
static Employee employeeExemplar(Exemplar());
extern Employee *employee = &employeeExemplar;
```

另一个选择是从同名样本的尾部分配，访问是通过全局指针来进行的：

```
extern Employee  *employee = new Employee(Exemplar());
```

这些定义能够放在自己的源代码文件中，或者放在相同的源代码文件中，样本类的实现细节在其中被履行。样本说明可以在全局头文件中进行发布。

样本对象用 Exemplar 对象作为参数来初始化，这种选择的构造函数习惯是从类模板中构建单一的原型对象。Exemplar 类不传送信息，在类中以及它本身都没有语义，它仅作为由原型创建的确定的（重载）构造函数。即只要 Exemplar 类具有良好的行为（操作）并具有一个构造函数，那么 Exemplar 对象做什么就都无关紧要了：

```
class Exemplar {
public;
   Exemplar()  {/*null*/}
};
```

make 成员函数能够在样本上被引用，以便创建新的对象来接替构造函数角色。在 C++中，构造函数和析构函数是“专用”的不像其他成员函数，在这些专用函数中，把发生在类本身的操作用发生在此类的对象上的操作来替代。make 成员函数像其他成员函数一样，除非按惯例要求从样本中建立新的对象。在正常情况下，make 返回一个指向样本拷贝的指针，作为它的返回值。所有新类对象都是从样本中用“克隆”的方法创建的。我们可以重载 make 操作，而自变量能够用于参数化结果对象的内容中（像变异的克隆一样）。每个 make 可以调用类似于特征的构造函数，通常在同一类的 private 或 protected 接口中能够找到。这里，我们通过引用构造了一个新的 Employee 对象：

```
class *smith = employee ->make("Smith", "9120784393");
```

对象的删除遵循传统的规范格式，使用该类的析构函数来构造样本；这个类的析构函数必须说明为 virtual。

现在，我们把这些思想组合到样本中，再现 Employee 的抽象。例 8.2 表示了这种类本身的说明，包括构造函数用类 Exemplar 参数去建立“父辈”样本对象。在 Employee 接口定义中包含有一个缺省的 make 操作，并且 make 操作用名字和雇员的标识号来创建新的 Employee 实例。这里，头文件包含了样本处理 employee 的说明。

例 8.2：Employee 样本的说明文件 Employee.h。

```
#include "Class.h"
#include "Hours.h"
#include < String.h >

typedef long EmployeeId;

class Employee:public Class {
public:
   Employee(Exemplar /*未使用*/) {}
   // make()函数取代了构造函数
   Class *make() { return new Employee; }
   Class *make (const char *name, EmployeeId id) {
```

```
        return new Employee (name, id);
    }
    long printPaycheck();
    void logTimeworked(Hours);
private:
    // 注意：构造函数是私有的，其含义是这个类的原始实例不能被建立。
    Employee():salary(0), vacationAllotted(0),
        vacationUsed(0), name(""), id(0)  {}
    Employee(const char *emp_name, EmployeeId emp_id):
        salary(0), vacationAllotted(0), vacationUsed(0)
        {
            name = emp_name; id = emp_id;
        }
    Dollars salary;
    Days vacationAllotted, vacationUsed;
    String name;
    EmployeeId id;
};

// 这个变量对 Employee 样本对象以全局方式处理
extern Class *employee;
```

为了从它的实现中获得各自样本接口的充分的好处，我们希望用户代码一点也不依赖于类 Employee 本身的说明。这就意味着，对于用户的 Employee 协议必须根据更一般的抽象，诸如 Class。在例 8.2 中，注意用 make 成员函数返回 Class 对象指针来替代例 8.1 中的 Employee 对象指针。由于在 Class 中复制 Employee 的操作是虚拟的，我们把在 Employee 本身的布局或私有成员函数中的变化与用户进行了隔离：

```
class Class {
public:
    virtual Class *make() = 0;
    virtual Class *make(const char*, EmployeeId) = 0;
    virtual long printPaycheck() = 0;
    virtual void logTimeWorked(Hours) = 0;
    ...    // 对其他导出类的纯虚拟函数
};
```

样本处理说明 Employee 需要从 Employee 说明本身分别加以管理，可能用在每个用户源代码文件中复制的方式来维护 Employee 成员函数的所有访问。为了使所有样本得到这种好处，一般地，类 Class 必须说明为所有样本成员函数的联合体（union），使之适合管理潜在的不符合需要变化的连锁反应的瓶颈。把成员函数说明放到层次结构的顶部和在 Class 自身层次下实施应用的访问之间进行权衡这部分内容在本章后面将加以讨论。

.c 文件仅包含样本本身（文件说明为静态，所以它保持该文件外部不可见），而对样本的外部可见性“处理”用 Class*来加以说明：

```
#include "Employee.h"
// 这是样本对象本身：创建了所有其他 Employee 对象的父辈
static Employee employeeExemplar(Exemplar());
```

```
// 这里，初始化全局的 Employee 样本处理。
Class *employee = &employeeExemplar;
```

另一种选择形式是：

```
#include "Employee.h"
Class *employee = new Employee(Exemplar());
```

如果在头文件中此成员函数没有作为内联（inline）来定义，则成员函数可能正好已经定义在这个文件当中。

程序使用样本与使用纯的相同抽象的基于类的实现方式有些不同（见例 8.3）。我们用在样本上执行 make 操作来替代类的 new 引用，这是未知的专用类型：它是作为 Class*来访问的。

例 8.3：具有样本的传统规范习惯用法的比较。

Class 代码：

```
#include "Employee.h"
int main()  {
    Employee *ted =
        new Employee(
          "ted", 2823763108
        );
    ...
    ted->logTimeWorked(8);
    ted->printPaycheck();
    delete ted;
}
```

Exemplar 代码：

```
#include "Employee.h"
int main()  {
    Class *ted =
      employee->make(
        "ted", 2823763108
      );
      ...
    ted->logTimeWorked(8);
    ted->printPaycheck();
    delete ted;
}
```

如果我们要求强类型检查，那么样本指针能够被说明，并且使用 Employee*，正如例 8.1 原先表示的那样。类型检查是以 Employee 用户和 Employee 的类接口之间的紧耦合为代价的。在这种情况下，不存在把样本处理说明放入头文件中的不利后果。更多的“类似 C++”的风格可以被采用，样本的作用域与静态公有类成员一样（见例 8.4）。该样本用 Employee::exemplar 来访问，以便替代应用代码中的 employeeExemplar：

```
Employee *aWorker =
          Employee::exemplar->make("Joe", 123456789);
```

例 8.4：样本源代码文件管理。

Employee.h 文件：

```
class Employee{
public:
    static Employee *exemplar;
    ...
};
```

Employee.c 文件：

```
Employee *Employee::exemplar = new
Employee(Exemplar());
```

当使用这种习惯用法时：当要在运行时改变类型的特征时，就要使用样本。例如，改变 Employee 样本的数据可能要改变所有现存的 Employee 对象的行为（类型特征）。对给定类型的所有对象来说，样本作为对象的管理程序也是有用的。例如，样本能够检查跟踪给定类的轨迹，或者能够管理给定类的资源：属性和功能。它们的主要用途是作为习惯用法的基础，这些内容将在本章的后面及第 9 章中加以描述。所有的目的是为了增加运行时的灵活性，减少变化的影响。

8.2 样本和类属构造函数：样本共用的习惯用法

样本能够被添加到C++程序设计的类属层上[2]，并且C++允许成员函数是虚拟的，“虚拟构造函数”在语言中毫无意义。但是，当建立的对象类型是构造函数参数或全局上下文的函数时，就需要引入虚拟构造函数。采用这里所描述的技术，样本就能接近这些语义。

如果有一些无名的数据，诸如名字和标识号，这些数据本身已包含了足够的信息，使Employee类能够标识专用的雇员类型，即，经理、在线工人（line worker）、库房工人等。这些数据可以来自于磁盘文件，或者由人工操作交互式输入。解释这些数据的专门知识存在于Employee类中。现在，我们要求建立一个对象，以适合雇员类型指定的数据。但是，因为建立对象的代码紧密地与编译时的对象类型联系在一起，又因为数据在运行之前是未知的，所以该类系统不能直接地获得这些信息。

惯用解的一种集合已经在第5章的“虚拟构造函数”一节中作过介绍。样本也是这些问题的一种自然解，并可以作为虚拟构造函数方法的概括。记得对象是由对样本对象进行make操作的样本构成的，并且make能够被重载，正如构造函数可以被重载一样。如果这些make成员函数仔细检查了它们的参数，那么它们就能够确定什么样的类是合适的，并安排返回指向合适类型或参数化对象的指针。

例如，假定公司的雇员角色是根据它们的I.D.来编码的，比如第一个字母为M的是经理，第一个字母为L的是在线工人，第一个字母为V的是副总裁等等。如果公共操作是由名字/I.D.号对构成Employee对象，那么我们就可以有一个像例8.5一样的组织安排。

例8.5： 一般的Employee构造函数。

```
class Class  {
public:
    virtual Class *make(Const char*, EmployeeID);
    ...
};
class Employee: public Class {
public:
    Class *make(Const char*, EmployeeID);
    ...
};
Employee *employeeExemplar = new Employee(Exemplar());
Class VicePresident: public Employee {
    ...
};
class LineWorker: public Employee {
    ...
};
Class *
Employee::make(const char *name, EmployeeID id)
{
    Class *retval = 0;
    switch (id.firstChar()) {
    case 'M':   retval = new Manager(name, id);
```

2 另一种替换方法将在第9章讨论。

```
                break;
        case 'V':   retval = new vicePresident(name, id);
                    break;
        ...
        }
        return retval;
    }
```

这里，所有 Employee 的导出类的消息都包含在该类的 make 操作中。Employee 在这种上下文中有两种角色：一种是作为把基类的导出类（即所有种类的雇员）的行为紧密联系在一起的基类；另一种作为所有 Employee 类型的部分来组织安排的样本。给出这种安排的程序为

```
int main()  {
    Employee *joe = employee->make("Joe",
                                   EmployeeID("M012345678"));
    ...
}
```

这样，就建立了一种“类型”Manager 的对象。

用单一的“母”样本收集与样本有关的处理是一种有用的习惯用法。这种习惯用法被称为**样本共用的习惯用法**：共用相关的样本可以通过用称为共用管理者的单一典型的方式紧密联系在一起。共用管理者为“母”函数服务，或者是更可靠的共用模拟，为所形成的对象的领导者角色服务。

当使用这种习惯用法时：当要求兼顾样本的好处、运行时灵活性以及“虚拟构造函数”的好处时，就要使用共用样本（习惯用法）来通信。

8.3 自主的类属构造函数

如果 Employee 样本理解所有的 Employee 导出类，那么上述例子就会运作得很好。但是，这却可能与演变的需要不相一致：我们无须用调节 Employee 样本的方式使之改变什么来添加导出类，从而违背 Employee 的抽象，我们应该做的是在具有合适的类中分配系统消息，以便遵循良好的面向对象设计技术。

要达到此目的就应按以下步骤执行：

（1）标识我们所要求处理的作为单一类属的样本（如以上雇员类）的相关对象的聚集。

（2）建立对所有这些类的成员函数可访问的全局列表，其中该列表的元素是指向这些类的对象的指针。

（3）对每个这样的类创建样本，并把这种样本的“运行记录（日志）”本身放入列表中。

（4）如果引用了不合适的参数，就对每个样本的 make 操作返回零（zero）指针。

（5）改变循环该列表的主要样本的 make 函数，对每个有它自己参数拷贝的样本重复应用 make，返回第一个非零返回值。

我们称这种方法为**自主的类属样本习惯用法**。每个样本有对其参数求值的消息；如果没有，其他样本就会有同样的机会。就好象所有市民聚集在政府广告牌前等待招工广告的发布一样：一旦有一个新的招工广告发布，市民们为了获得这个新的广告所发布的工作机会就按要求在此时排成一行。第一个人就会首先获得此工作的机会。

本章中最后介绍的例子就是应用这种方法的单一结构的语法分析程序。样本对象是作为每个Atom 导出类（Number，Name，Punct，Oper 等）由全局范围的静态变量创建的。每个样本对象的实例化导致了基类构造函数 Atom::Atom(Exemplar)的被引用；这个构造函数放在每个样本对象的链接列表 list 上，Atom 的静态成员是：

```
Atom(Exemplar /*未使用*/) {
    next = list; list = this;
}
```

Atom 类从实例化观点出发，它代表了其所有的导出类。包的用户仅构造了新的 Atom，即它们一般不直接构建 Number 对象和 Name 对象。Atom 成员函数 make(String&)把它的参数依次传送到每个对象列表的 make 操作中。第一个类在它的类型和 String 开始模式之间匹配查找，并从它的 make 操作中返回非零值：

```
virtual Atom *make(String &s) {
    Atom *retval = 0;
    extern Atom *atomExemplar;
    if(this ! = atomExemplar) {        // 避免无限地递归
        for (Atom *a = list; a; a = a->next) {
            if (retval = a->make(s)) break;
        }
    }
    return retval;
}
```

每个 make 操作在它的类构造函数的结果上传递，除非这个构造函数设定一个错误标志，以表示不能作为它自己的类型而被输入。用导出类 make 成员函数的方法返回第一个非零值是通过采用 Atom::make 作为对用户返回值的方法来传递的。

注意，对这个例子来说，导出类样本中放在列表中的次序是很重要的，因为在语法分析中次序会影响到词法上下文的语义。样本在试图匹配稍短的字符串之前，应当寻找匹配尽可能长的子串。

样本的优点是摒弃了类的许多特征。只需了解类对象的内部（私有成员）的工作的地方是新对象创建的地方，并在类成员函数内部。因为 new 不再用于建立对象（而由 make 来创建），所以就不需要类接口处处了解对象现在是否存在这一问题。现在这种消息被嵌入到每个对象的 make 函数。这就意味着类结构的消息又能够限定在该类的本身之中；类的用户则通过高层样本机制而相互作用，我们有如下描述。

当使用这种习惯用法时：当要完成在样本中避免类消息集中的“虚拟构造函数”的语义机制时，就要使用这种习惯用法。对象初始化的消息能够保留分布在跨越所有参与类中。其应用的例子包括语法分析和扫描分析程序。

8.4 抽象基样本

采用上面给出的这种方法，对象相互之间怎么联络呢？对象的通信仍然遵循 C++的约定，通过对象成员函数（即，我们不使用消息或事务或任何外部处理）进行。任何样本对象的用户必须有一个在编译时的样本类接口（特征）的说明。由于大多数样本对象的类接口驻留在它们的源代码文件

中，那么这种特征又来自于何方呢？

回答是来自于使用虚拟函数支持的高层体系结构的抽象当中，这种抽象被称为**抽象基样本**。单个类实现从全局使用的较高次序类中导出在全局头文件中的说明。这些高层次类落入到浅层次结构中，但在数量上是很少的。在最高点是 Class 类，抽象基类（如图 8.1 所示）的特征反映出系统中大多数类中的大部分操作：

```
class Class {
public:
   virtual Class *make() = 0;
   virtual Class *make(Const char*, EmployeeID) = 0;
   virtual long printPaycheck();
   virtual void logTimeWorked();
   virtual ~Class();
   ...
};
typedef Class *Classp;
```

即，我们从相关类的组中聚集每个成员函数说明，把它们合并到 Class 特征中。作为一个抽象基类，Class 是不能够被实例化的。但是，Class*（或它的等价的类型定义 Classp）能够引用 Class 的亲子关系类的任何对象。

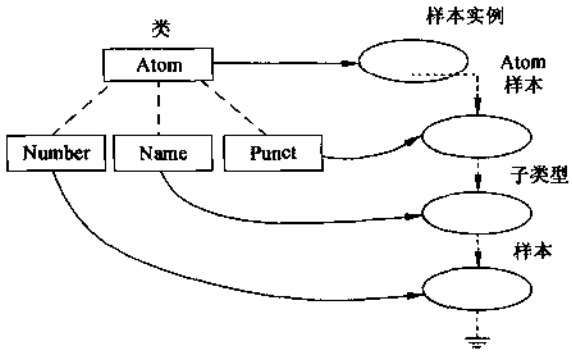


图 8.1 Atom 类的独立类属样本

所有对新对象的应用要求都用 Atom 样本实例来处理，其中用它的从属类的样本处理应用要求。在列表中第一个样本能够分析语法串 Atom::make，以便获得这样的处理，并且产生出它自己的类对象。

注意，在这种方法中，降低了对分类机制中的类的使用。把所有成员函数引入到基类 Class 中，就能够允许通过 Classp 指针访问任何对象；就好像每个对象为类 Class 一样。程序员只用到几个类，这几个类是为所有系统对象的接口服务的；这些类在浅继承层次结构的顶部，这种层次结构的底部是宽的，而在顶部是窄的。所以函数（而不是类）就成为了主要设计的关键点。但是，这些函数在许多 Class 对象构成的操作中仍然是多态的。这是一种使用 C++的“从里到外调节”的继承方法，称为**包含多态性**（inclusion polymor-phism）。我们认为，这种构成更多的依赖于参数化多态性：首先定义多态函数，并把从所有应用类中导出的子类收集到抽象基样本中。这类似于从设计观点出发的 C++函数模板的表示（见第 7 章），在这种类的结构中由宽的、浅层次的结构所组成。这里，我

们实现了函数模板的无源代码重用，但我们所获得的模板却不能提供运行时多态性。抽象基样本和更传统的基于类的方法之间的差别不在于结构，而在于着重点。无论是成员函数运行时标识，还是抽象基样本都存在着把类层次结构的分解归并到大型接口的单一结构的类中的问题。

当这些类属于一个层次结构时，说不定就可能是它们的样本实例。我们需要收集用户的类（如Employee 和 Atom），以便尽可能地在抽象最高层次上聚集运作，这就是为什么成员函数要被推到离层次结构顶部较近的类的原因。每种应用都有它自己的抽象基样本；对语法分析程序来说，抽象基样本是 Atom；而对工资单系统来说，Employee 是属于几个抽象基样本之间的。当然，它可能让用户使用深度类（需要添加规格说明）来访问各个层上的相同的层次结构。但是，这就使得用户的代码依赖于低层类的接口，并强制这样的用户到#include 的另一个头文件中去访问它的特性。

这样就导致了重要的设计权衡的出现。一种极端的情况是“胖”抽象基类在层次结构的顶部，它可被所有成员函数引用。这样的层次结构用户的接口必须只与单一的抽象基类有关。另外一种情况更具有传统的层次结构，其中特殊的成员函数仅出现在有关的导出类中。虽然抽象基样本在层次结构中创建的所有对象都能引用公共操作，但是低层的类接口必须对所有可使用的操作可见。权衡是在单一的精心制作的抽象和大量的结构单一的抽象之间进行的。这样的权衡应着眼于两个方面：一方面是希望软件的演变容易；另一方面是软件的演变尽可能少地重新编译。

在 Class 上的操作都是虚拟的，并且大多数是纯虚拟的。即它们只是作为位置占位符来出现，并且是没有体（body）的。在 Class 的虚拟操作之间定义的是 make 成员函数，这就意味着，如果一个程序有一个指向某个原型对象的指针 *p*，而且如果使用 Class.h 来编译，那么它能够使用 p->make 获得一个指向新的对象从原型中克隆的指针。典型的虚拟 make 操作调用它自己的构造函数，以便建立新的对象。

如何编写一段代码以获得指向合适的原型对象的指针呢？如上所述，大多数样本抛弃了在全局变量中指向它们自己的指针；仅当调用相应的 X::X(Exemplar)构造函数时，指针才会建立。用命名服务对象机制来整体地对命名这些对象进行选择，这种机制是由每个最新构成的原型日志本身来完成的。

小结：每个类和处理它的样本对象的代码同在它们自己的源代码文件中，只有全局输出的信息是一个处理全局样本的集合，而少数几个类的接口说明为它们所有的导出类接口的联合。这些抽象样本有一些在 C++头文件中定义的接口，而这个头文件是用任何程序员所希望的#include 来构造的，或者是用这些类型的对象所组成的。

8.5 关于样本习惯用法框架

至此，我们已把属于此类的信息很好地分离出来，并且当添加新的导出类时，我们已经解决了违反基类封装性的问题。但是，我们仍存在一个重要的问题：在给定的样本体中，把一个新的成员函数添加到任意一个类组一起必须被反映到某个公共父类中（通过这些对象的用户来访问最低的一个）。下面简单介绍更改影响的分析，使得在大多数 C++实现中所有这些类的代码无须重新编译。

一种解决方法是通过另一种方式来折衷。这种方式假设对每个类有一个成员函数（命名为 doit），

把由命令表示的细节操作当做参数，并对这种操作列出自变量表。最简单的类接口的花费在管理命令的全局列表上，其中必须把每次更新的语义接口添加到系统中。这种作法很像命名为 doit 的弱语义成员函数的使用，我们建议从构成原因或弱映像的应用实体中获取语义，并且撤销了在第 6.5 节中讲过的注意事项。不管怎样，把新的语义操作添加到该接口就不再产生新增加的成员函数——就好像使用合适的由导出类解释的新的参数 doit 一样——这样，重编译的更改影响要比以前少得多。成员函数不再表示应用的抽象行为；映像表由 doit 使用。类描述“胖接口”，不是因为基类有一些导出类不需要的函数，而是因为成员函数参数能够包含函数、活动、检索或其他映像的任意宽的范围。这些操作使人想起人工智能所用过的槽和框架。槽是信息段的位置占位符；对象槽（状态变量）填入得越多，就表示完成对象抽象表示的描述越多。

实际上，在我们的接口上采用的语义层次就是上述的“doit”层次。在本章末尾介绍时，基于样本的例子就精心制作了上面提到的 Employee。在这个例子中，类 Class 用于访问在不同的领取薪金的雇用种类中各种类型的对象。Employee 由 Class 导出，而 Employee 有多个导出类，如 VicePresident，Manager 和其他协同角色。

在本章末尾的样本例子中，处理包括填入的槽名和被检索的数值的所有对象，命名了的按钮触发执行某些参数上下文中的一些动作。因此，在所有对象上的基本操作是 at，它通过槽来执行相关的检索；atPut 执行相关的存储；而函数 getNext 和 putNext 用于处理一系列聚集的对象（例如，按顺序扫描其槽）。atPut 也可以对刚按下的“按钮”的副作用作出评价，其中所提供的参数是相关活动的数据，而不是把某件事情装入槽内；printPaycheck 命令是一个例子。这里，关键字段是枚举常数；用 6.5 节中关键字串展示的符号方法效率较低。

因为多态性是在这些访问函数层上来处理的，而不是在虚拟函数提供的成员函数层上处理，所以某些多态和继承的语义必须在这些类本身显式编码。这种结果容易得到。在 VicePresident::atPut 中，我们看到对这些指令的缺省的 case 语句，对这些语句，样本是不承认的。条件的语义是把操作转发到 VicePresident 的基类，命名为 Employee，其动作是作为在 VicePresident 上执行 atPut 操作子集的委托，所以此种行为更像在 Smalltalk 看到的方法。此方法要求过滤出寻找能够处理“消息”的类的继承链。这种人工管理的多态性类似于 C++中所提供的虚拟函数，所不同的是不存在编译时的类型检测，所以例外可能在运行时产生。这里，未处理的访问函数信号要求返回 null；更极端的动作（如产生例外）完全可以用编码来替代。

函数与槽之间的折衷相对于灵活性与方便性之间的折衷，它们之间的关系是可变的。增加新的样本或成员函数一点也不用管理。使用抽象样本，新类可以加到程序中，就好像用编译方法把新类型和增量装入到已有的代码中一样。某些环境可以允许将链接-编辑新对象代码增量装入到已有的模块中（在 UNIX 下，是 a.out 文件）。在用增量运行装入的连续-运行系统中，新的样本可动态地合并到运行程序中（参见第 9 章）。例如，如果增量语法分析程序（本章开始部分已作过描述）是在线预约系统的查询语言处理程序的一部分，我们就能使用操作系统把新的语言构造类装入到运行的、操作的系统中。在样本的习惯用法中，软件装入机制把新的样本添加到样本列表中，并用样本管理来维护。在此列表中的位置将新的样本链接到系统中，新的软件功能立刻变成可变的，并且被安装，同时，它也不影响已有的软件。

样本习惯用法不是 C++所“固有”的，这是一种缺憾，给大型软件项目的培训带来一定的困难，这就要求新的开发工具来有效地管理此种风格的程序。另一个副作用是，现在很少能确保在编译时

此种操作只在合适类型的对象上引用；类的实现必须是在不希望的要求被执行时，用合适的动作响应。性能也会比用编译时的联编（风格）要差些。

当使用这种习惯用法时：当运行时的灵活性比编译时的类型检测优点更重要时，就要使用这种习惯用法。使用样本能降低重新编译时更改所产生的影响，以支持原型和探索程序设计中的快速迭代。

8.6 相关的表示

在 6.4 节中，对象和类联系的表示已作过描述。那种表示基于类和对象的“本机”C++模型，是应该扩充样本推荐的模型，以便处理所带来的丰富的联系。

大部分的更改比基本含义更多，也更丰富。例如，IS-A 现在可能作为实例之间的关系：任何对象从作为另一个对象的克隆出发，对原始的对象存在着 IS-A 关系。克隆可能同时与其他附加修饰语有关，向这些对象转发某些操作，以更改这些对象的功能。如果克隆的特征保留原始的基本语义（像第 6 章所描述的那样），那么 IS-A 关系就仍会存在。

如果使用样本习惯用法，类只能是二等“公民”。一方面，HAS-A 关系不再有类的含义；另一方面，对象的含义有着不同的微妙的含义：事事都要看它的实例。CREATES-A 关系使类和对象之间的暧昧特征失去了；现在所要做的就是看它的对象了。

当然，类仍然是主角，并且我们可以采用所希望的给定的表示。样本替换类，而 C++类的样本是这些“类代理”的类。几种语言调用这个类的元类（metaclass）的类，所以每个样本实例对 C++类来说都称为具有 HAS-METACLASS 关系。

因为样本在角色中经常使用与类相关的对象实例，所以获取程序的语义要求在单一展示中采用对象和类关系的混合表示。图 8.2 是 Atom 例子的这种类型的图；这是图 8.1 的“形式化”的描述，使用了第 6 章所介绍的 Booch 表示，在简单应用的上下文中为

```
extern Atom  *atom;
Expr parse(String s)  {
    ...
    Atom *lhs = (*atom)->make(s);
    Atom *oper = (*atom)->make(s);
    Atom *rhs = (*atom)->make(s);
    Expr expr = (*exprExemplar)->make(lhs, oper, rhs);
    ...
    return expr;
}
```

这里，Oper，Punct，Number 和 Name 是该应用的元类。对样本对象 operInstance，punctInstance，numberInstance 和 nameInstance，它们各自的元类关系用它们之间的元类的箭头来表示。不管怎样，Oper，Punct，Number 和 Name 用 C++类来实现，而这里它们是使用类的图标（黑方块点）描述。该应用的“类”（operInstance，nameInstance 和其他实例）是样本，在 C++中是对象，并反映到图 8.2 中。应用对象（oper，rhs 和 lhs）与它们的样本之间的对象所建立的关系用图中的虚线来表示特别的关系。

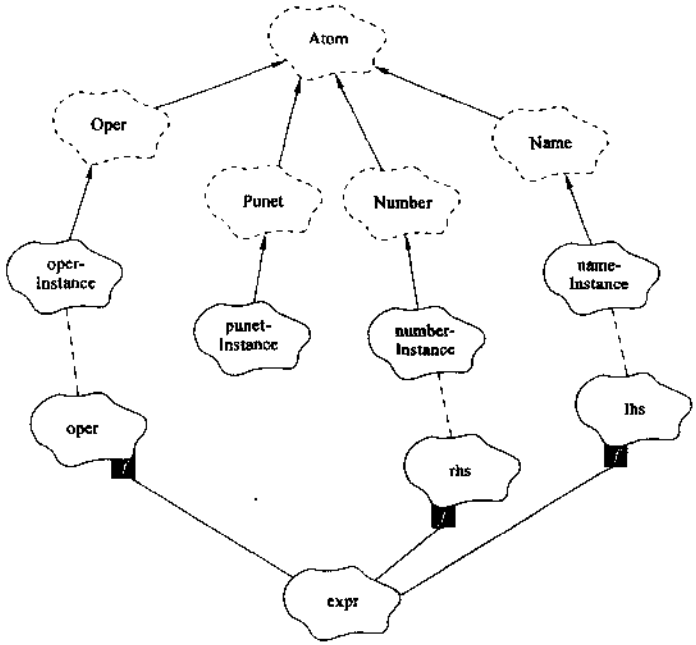


图 8.2 混合的对象/类图

8.7 样本和程序的管理

这种风格导致程序结构具有很少的头文件。典型地，每个类有一个源文件；该类的说明在源文件本身，而不在头文件。一个类的单一源文件存放的地方太大了；在这些实例中，类说明是放在局部头文件中，而不是全局头文件中。一个类不经常访问的细节存放的地方需要用到某些其他类的对象，强制某些类说明在全局头文件中。所以，类说明出现在 3 个地方：在模块源文件中；在局部头文件中和在全局头文件中。

特别值得注意的是，必须给定的枚举值就作为在框架样本习惯用法中的指令。因为特殊的具体的子对象功能不可能出现在抽象基类的一般化接口中，所以子协议必须在这些定义内用抽象基类接口建立。该协议必须限定在编译时是有效的。为了避免在这些协议中发生冲突，它们必须有中心的管理机制。这样就可能导致长的、编译时熟知的“消息类型”列表，即在某种意义上等价于全局数据。但是，因为这样的消息类型基本上是枚举类型，它们可以按线性排序，并在结尾添加新的类型，对其余的程序无论是在语义上还是在重新编译时都不会有严重的影响。

如果从效率方面做一下选择的话，则消息类型的索引应能约束运行时相继产生的整数值。因为样本需要消息类型，所以它们能够建立新的 MessageType 对象，并在它们自己的名字空间中与其名字捆绑在一起：

```
class MessageType  {
public:
   MessageType():typeField(nextType ++)    {}
```

```
    operator int()  {return typeField; }
    int operator == (MessageType m)  {
        return typeField == m.typeField;
    }
private:
    Static int nextType;
    int typeField;
};
int MessageType::nextType = 0;
class AssociativeArray  {
public:
    AssociativeArray()  ...
    void atPut(void *element, MessageType key)  {
        if  (key == Size)  sizeFrame = *((int*)element);
        else if  ...
    }
    void *at(MessageType key) {
        ...
    }
    static MessageType size, print, ...
private:
    int sizeFrame, ...
};
static MessageType AssoclativeArray::size,
          AssociativeArray::print ...
int main()  {
    AssociativeArray a;
    MessageType AssociativeArrayExample,
        CirclistExample;
    a.atPut((void*)233, AssociativeArrayExample);
    ...
    int  Size = a.at(a.size);
    ...
}
```

这种方法比用 enums 方法的效率低些，但确实比使用 String 方法的效率要高得多。

练　习

8.1 修改本章末尾的单一结构的语法分析程序，以使 Atom 构造函数能够在同一串上引用多次，其开销在识别记号（token）和把记号转换到对象之中。

8.2 编写抽象基样本应用的名字服务器，比如单一结构的语法分析程序。每个样本应有名字服务器的自身日志（何时？）。编写 lookup(const char*)成员函数，为名字服务器提供名字到样本的映像。那么应当在什么地方保存样本符号名字的信息？

8.3 为了使上述名字服务器对任何给定样本的符号查询有效，仅当 lookup 第一次用给定的串时才被调用（提示：允许用户对 lookup 扫描 String 迭代程序对象。记住，lookup 的相关的交叉调用）。

8.4 完成下面基于框架的样本例子：

基于样本的单一语法分析程序

这是一个单一语法分析程序的样本的实现（第5章推荐的例子）。

```
class Exemplar { public: Exemplar()  {} };
class Atom {
protected:
   static Atom *list;
   Atom *next;
   static char errFlag;
public:    // 所有Atom的特性的联合
   Atom(Exemplar)  {next = list; list = this;}
   virtual Atom *make(String &s)  {
      Atom *retval = 0;
      extern Atom *atomExemplar;
      for (Atom *a = list; a; a = a->next)  {
         if (a ! = atomExemplar)  {
            if (retval = a->make(s)) break;
         }
      }
      return retval;
   }
   Atom()  {next = 0:errFlag = 0;}
   virtual ~Atom()  {}
   virtual unsigned long value()  {return 0; }
   virtual String name()  {return String ("error");}
   virtual operator char()  {return 0;}
};

Atom* Atom::list = 0;
static Atom atomInstance(Exemplar());
Atom *atomExemplar = &atomInstance;

class Number:public Atom {
public:
   Number(Exemplar a):Atom (a) {}
   Number(String &s):Atom () {
      sum = 0;
      for(int i = 0; s[i] >= '0' && s[i] <= '9'; i + +)  {
         sum = (sum * 10) + s[i] - '0';
      }
      char c = s[i]; errFlag = !(ispunct(c) | | isspace(c));
   }
   Number(Number &n)  { sum = n.value(); }
   Atom *make(String &s) {
      Atom *retval = new Number(s);
      if (errFlag) { delete retval; retval = 0; }
      return retval;
   }
   ~Number()  {}
   unsigned long value() const { return sum; }
```

```
private:
   unsigned long sum;
};

static Number numberInstance(Exemplar());
class Name:public Atom {
public:
   Name(Exemplar a):Atom(a) {}
   Name(String &s):Atom()  {
      for(int i = 0; s[i] >= 'a' && s[i] <= 'z'; i + +)
         n = n+s(i, 1);
      // 如果这不是一个名字，通过 errFlag 告诉调用者
      if (!isalnum(s [i])) errFlag + +;
   }
   Atom *make(String &s) {
      Atom *retval = new Name(s);
      if (errFlag)  {delete retval; retval = 0; }
      return retval;
   }
   Name(Name &m) { n = m.name(); }
   ~Name() {}
   String name() {return n;}
private:
   String n;
};
static Name nameInstance(Exemplar());
class Punct:public Atom {
public:
   Punct(Exemplar a):Atom(a) {}
   Punct(String &s):Atom()   {
      if (!ispunct(s[0])) errFlag + +;
      s = s(1, s.length() -1);
   }
   Atom *make(String &s)   {
      Atom *retval = new Punct(s);
      if(!errFlag)  {delete retval; retval = 0; }
      return retval;
   }
   Punct(Punct &p) { c = char(p); }
   operator char() {return c;}
   ~Punct() {}
private:
   char c;
};
static Punct punctInstance(Exemplar());
class Oper:public Atom {
   ...
};
```

基于框架的样本

这个代码表示了各种各样类的简单程序框架，诸如 Employee 和 Dollars 共享公共的抽象基样本 Class。Class 类是在继承层次结构中类下方的“胖接口”。

```
extern class Class *dollarsExemplar;     // 转发引用
class Class {
public:
   enum Directive {PrintPaycheck, TimeWorked, GiveRaise,
                   Salary, Stock, ToString, Value };
   virtual Class *atPut(Directive key, Class *arg) = 0;
   virtual Class *at(Directive key) = 0;
   virtual Class *getNext() = 0;
   virtual Class *putNext (Class*) = 0;
   virtual Class *make(Class *param) = 0;
   virtual Class *make(int = 0, int = 0, int = 0, int = 0);
   virtual int toInt() = 0;
};

class Employee:public Class {
public:
   Class *atPut(Directive key, Class *arg)  {
      switch(key)  {
      case Salary   salary = arg->toInt(); return this;
      case Print:   cerr << *this; return this;
      default:      return 0;
      }
   }
   Class *at(Directive key) {
      switch(key) {
      case Salary:  return dollarsExemplar->make(salary);
      ...
      default:      return 0;
      }
   }
   int toInt() { return salary; }
   Class *make(int = 0, int=0, int=0, int=0) = 0;
   Class *make(Class param) = 0;
   Class *getNext() = 0;
   Class *putNext(Class* ) = 0;
protected:
   Dollars salary;
   Days vacationAllotted, vacationUsed;
};
class vicePresident:public Employee {
public:
   Class *atPut(Directive key, Class *arg) {
      switch(key) {
      case GiveRaise:salary = arg->toInt() + salary;
                     stockOptions* = 1.05;
                     break;
      default:       return Employee::atPut(key, arg);
                     break;
      }
      return this;
   }
   Class *at(Directive key) {
      switch(key) {
```

```
        case Stock:return dollarsExemplar->make (stockOptions);
        ...
        default:   return Employee::at(key, arg);
        }
    Class *make(Class *param) {
        name = param->getNext() ->at(ToString);
        salary = (int)param->getNext()->at(ToInt);
        stock = (int)param->getNext()->at(ToInt);
    }
    Class *make(int i=0, int j=0, iht k=0, iht l=0) {
        Class * n = new VicePresident;
        n->salary = i; n->stockOptions = j; returnn;
    }
    Class *getNext () {return 0;}
    Class *putNext (Class* ) { return 0;
    VicePresident (Exemplar) {}
private:
    Dollars stockOptions;
};

class *vpExemplar = (Class*)new VicePresident(Exemplar());

class Dollars:public Class {
public:
    Class *atPut(Directive key, Class *arg) {
        switch(key) {
        case Value:  rep = (int)arg->at(ToInt); return this;
        default:    return Class::atPut(key, arg);
        }
    }
    Class *at(Directive key) {
        switch(key) {
        case Value: return integerExemplar->make((Class* ) rep);
        default:    return Class::atPut (key, arg);
        }
    }
    int toInt() {return rep;}
    Class *make(int i=0, int j=0, int k=0, int l=0) {
        Class * h = new Dollars;
        n->rep = i; returnn;
    }
    Class *make(Class *param) {
        Class * n = new Dollars;
        n->rep = (int)param;
        return n;
    }
    Class *getNext() {return 0;}
    Class *putNext(Class* ) {return 0;}
    Dollars(Exemplar) {}
private:
    int rep;
};

class *dollarsExemplar = (Class* ) new Dollars(Exemplar());
```

第9章

在C++中模拟符号语言的风格

C++提供了定义抽象数据类型（ADT）的基本机制，并采用ADT进行面向对象程序设计。但是，高层面向对象程序设计语言（像Smalltalk和CLOS）的灵活性，在语言中要达到像C++语言那样紧密地与C联系在一起将是很困难的。在C和C++中，变量名紧密地与表示对象存储的地址捆绑在一起，以替代标号。标号可以从对象上取下并贴在所希望的另一个对象上。这种强联编，使编译能够确保给定的变量总是与给定类型的对象在一起。编译用这种方式产生有效的代码。效率和类型安全性取决于运行时灵活性所付出的代价：例如，变量用float说明绝不能作为运行时Complex对象的地址，虽然这两种类型在行为上是兼容的。

Smalltalk和大多数基于Lisp的面向对象语言提供了两种特征，这两种特征均基于变量和对象之间的松散耦合，C++不直接支持，但可以用适合程序设计风格的习惯用法来表达。第一个特征是自动内存管理（引用计数和垃圾收集）。变量是在对象标号的符号语言中，对象的生命周期依赖于指定这些对象的变量。符号语言环境使用基于松散名字-值联编这样一种专门技术自动地清除不再使用的对象。C++程序通过指针模拟这种用地址对象构成的松散联编，而且这种附加的间接层次可用于自动内存管理，其详细的描述在3.5节和3.6节已给出。

高层面向对象程序设计语言的第二个特征是高层的多态性：习惯用法提供了这样的多态性（见5.5节）。这样扩充的多态性支持更柔性的软件体系结构；对象变成更松散的耦合，于是可以很容易地、独立地维护这些对象。

自动存储管理和扩充的多态性基于Lisp系统，Smalltalk以及在传统符号语言系统中的其他面向对象程序设计语言的两个主要特点。因为可以在C++中通过建立习惯用法来模拟这些特点，所以我们能够使C++程序员获得这种灵活性。这种灵活性是需要付出代价的：运行时性能的损失和存储开销需要与这种灵活性进行权衡。当使用这些习惯用法时，由编译类型系统检测预先的类型错误推迟到运行时才做，并且依赖于提供类型检验代码的完整性。这种权衡的规则不是一成不变的，但设计的经验和实践可以帮助软件工程师从本章或其他章节的习惯用法中进行选择，以便适应现有的应用。

本章把3种习惯用法组合起来。首先要把前几章的思想集成起来，通过减少变化的影响来支持迭代开发的风格。传统的规范格式提供了减少变化影响并作为其他两种习惯用法的基础。第二种习惯用法通过单一运行时环境可以提供增量运行时程序的更新。第三种习惯用法是一种自动清除和回收不再使用对象的技术。每种习惯用法既可以独立地使用，也可以把几种组合起来使用。

第二种习惯用法需要专门的经验和合适的平台：它依赖于C++类数据表示的实现细节。这里，实现的描述基于ANSI的C++标准语言版本。目前，C++遵循国际标准ISO/IEC FDIS 14882：X3j16/97-14882。当然，许多可移植的开发环境还是依赖于开发商。

本章的重点不应是解释作为替代 Smalltalk 和基于 Lisp 的面向对象程序设计环境这方面的技术。此外，在支持增量开发的灵活性上，符号语言具有增量开发所支撑的传统的强有力的环境。这里，灵活性也可以采用 C++的几个步骤来描述，但是需要附加代码的开销，因而潜在地减少了运行时的性能。本章试图促进支持增量式 C++开发的思想，并提供更新系统的灵活性以支持连续运行的应用，也可以作为代码自动生成工具的模型。

9.1 增量式 C++开发

为什么要进行增量式 C++开发呢？因为我们需要快速演变（系统），并且要确保不丢失“驱动测试”的新代码的进程，这是我们一贯的意图。快速迭代是一种改善系统的结构并且根据新的或变化的需求探索新的应用行为的重要技术。增量式演变的开销必须很低，从而使这样的迭代有效。

渐进性和面向对象设计

增量式开发原本就适合面向对象设计。对类的内部实现细节的封装就是普通的迭代单位；在继承层次结构中对所有类的公共协议的实施，减轻了新类（作为叶子）所添加的开销。所有这些都能够作为 C++设计的辅助手段，但重编译和重装入的时间可能比 Smalltalk 和基于 Lisp 的系统要慢。虽然强有力的 C++程序设计环境正在出现与传统软件技术决裂的态势，以提供增量式开发模式，但还是不够彻底，仍然存在着这样的技术平台是不可使用的问题。例如，在可交付使用的应用平台中，许多这种灵活性是我们所希望的，但作为嵌入式系统的运行却超越了任何“友好”的操作系统和传统支撑环境的范围。在本章中，我们探索的增量式方法比所希望发现的集成增量式 C++开发环境的能力要低一些，但却能够支持许多平台和环境的增量式开发。

减少编译的开销

增量式 C++开发的第一步是减少重编译的开销，一个有效的方式就是减少重编译的要求。我们知道在 C++变量、类型和约束存储之间存在着强编译时的关联，如果程序的演变改变了变量类型的特征或置换了地址，那么代码就会涉及到必须被重新连接和重新编译的名字。改变一个符号可能会产生邻近符号位置的偏移，也可能会导致必须重新编译所有的引用。例如，任何类的接口的变化就可能触发所有引用接口的那部分代码的重新编译。

大多数避免编译的方法是基于添加一个符号引用的间接层次。信封/信件习惯用法（见 5.5 节）和它的派生（诸如第 8 章的样本习惯用法）是这种方法的例子。本章所介绍的习惯用法主要是从样本方法中提取的。

对具有最小编译的变更的灵活性所带来的性能开销，来自于运行时附加的间接层。大多数符号程序设计语言比本机 C++语言在编译时的类型安全性方面有所欠缺。在开发项目中，当使用这种习惯用法时，对这两者的利弊权衡必须小心。

减少连接编辑和装入的开销

增量式开发的第二步是减少连接编辑和装入所需要的时间。通常连接编辑这一步需要建立可执行程序文件（重新定位编译对象文件），并且装入是一个把这些文件的代码装进可执行内存的操作。某些系统把这两步合并，而大多数系统在某种程度上把符号——地址装订在一起。

连接和装入的效率特别是对大型系统开发来讲是重要的，需要特别关注的，因为在大型系统开发中，面向对象技术具有可调整的博奕结果。大多数微型计算机和微处理程序操作系统的惯例在增量连接和装入方面不是很强。但是，许多新的 UNIX 版本和其他系统版本有增量式连接约定。增量式连接在小对象模块中的结果，所提供的实质性的周转比整个连接编辑要快。

即使连接编辑是快速的，重新装入仍然是一个瓶颈。如果系统的初始化所花费的时间过长，那么即使增量式连接-编辑改变现象发生，在每次迭代时仍然需要等待的时间。但是，如果代码能够递增式地装入到初始化运行系统，那么就存在着潜在的细微改变能够快速地周转问题。

快速迭代的展望

快速迭代对早期体系结构的研究十分有效，它放弃了原型的建立以改善对应用的理解。快速迭代能够作为产品开发的规范，但仅在稳定的体系结构所构造的限制之内。快速迭代对系统的向上兼容的进化是一种潜在的成功的开发的技术，但管理要适当。脱离主要接口进行迭代开发会招来灾难，从而加剧了平均信息量，破坏了系统结构。

9.2 符号规范格式

符号习惯用法给我们（3.1 节已介绍过）传统的规范格式提供了可以选择的方法。在 C++中，模拟符号范型是“非正统”的，或者至少在它自己的传统层次上要求具有它本身的约定和格式。用这种可替换的 C++类的传统格式，C++代码就能够模型化一般的符号程序设计语言的许多特性，但却失去了直接用传统规范格式能够实现的紧凑表示。

当使用这种习惯用法时：希望 C 和 C++在文化上兼容时就要使用这种习惯用法，但这需要符号语言环境的灵活性和渐进性。这种习惯用法也可以作为符号程序设计环境和 C++环境之间的接口的框架使用。这里，所描述的风格和习惯用法可用于建立 C++原型环境，提供了在应用软件中以下的一些程序设计约定。最后，习惯用法可在连续运行系统中使用，以支持运行时更新和长期演变。

在前几章中，传统的规范格式建立了内存管理和多态性概念，把这些概念加以扩充就可以支持添加的渐进性。传统的规范格式主要包括：

- 基于引用计数信件类的自动内存管理（见 3.5 节）；
- 取消终端用户对对象指针的操纵，尽管每个对象使用指针时均有其灵活性（见 3.5 节）；
- 依赖于运行时和装入时间灵活性的虚拟函数；
- 使用样本习惯用法（见第 8 章）。

符号规范格式的主要构造块是基类小的聚集，它是用于构造应用的信封和信件类。这些类的说明置入全局头文件 k.h 中，如例 9.1 所示。例 9.1 中有两个类：一个是 Top，作为所有信封类的基类；另一个是 Thing，作为所有信件类的基类。Thing 类本身实际上是由 Top 派生的，在许多符号程序设计环境中按照模型化根（root）基类来建立整个系统的一致性。这里，Top 和 Thing 所做的事情有点模仿 Smalltalk 中的 Object，Class 和 Behavior 角色，虽然不直接在两个方法之间映像是显而易见的，但在某些方面仍然是有用的。

这些都提供了我们所希望采用的符号语言环境的灵活性、内存管理支持、运行更新支持和松散类型模型。为此，下面两小节将详细地加以描述。

例 9.1：k.h 头文件。

```
// include file k.h
class Top {
public:
    // 这个类的对象没有数据，除非由编译提供了_vptr，
    // 从这个类派生的所有类保证_vptr 总是在对象中的第一
    // 个元素。在许多实现中都是这种结果，而某些实现需要采
    // 用不同的机制去访问 vtbl，只有符号习惯用法的动态装入
    // 方面依赖于这种性质
    virtual ~Top()  { /* 空的 */ }
    virtual Top *type()  { return this; }
    // 对实时更新清除删除是公有的。
    static void operator delete(void *p)  {
        ::operator delete(p);
    }
protected:
    Top()  { /* 空的 */ }
    static void *operator new(size_t 1) {
        return ::operator new(1);
    }
};
typedef unsigned long REF_TYPE;
class Thing:public Top {
    // 所有"rep"域都从 Thing 派生；它定义了所有信件类的规范格式。
public:
    Thing()  { /* 空的 */ }
    virtual REF_TYPE deref()  {          // 非块形 ref 计数
        return --refCountVal;
    }
    virtual REF_TYPE ref()  {            // 块形 ref 计数
        return  ++refCountVal;
    }
    virtual Thing *cutover(Thing*);      // 类更新函数
    virtual  ~Thing()  { /* 空的 */ }    // 析构函数
private:
    REF_TYPE refcountVal, updateCountVal;
};
```

Top 类

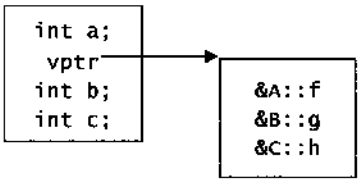
Top 类位于系统类层次结构的最高点，所有类都从 Top 派生出其特性。Top 没有显式的数据；但是，大多数 C++编译程序提供一个隐式数据成员，以支持 C++虚拟函数的调度。这种数据成员称为 vptr，并且它指向虚拟函数实体表，有时还称为 vtbl。类 Top 有一个虚拟函数，以确保这样的指针存在。

C++不同的实现可以实现不同的虚拟函数，但大多数在相同的主题上是变化的：

```
class A {
public:
    int a;
```

```
    virtual void f(int);
    virtual void g(int);
    virtual void h(int);
};

class B: public A {
public:
    int b;
    void g(int);
};

class C: public B {
public:
    int c;
    void h(int);
};
```

给出这些说明，类C的一个对象有点像在内存中有关说明：



如果最高点类没有数据，那么vptr就是所有感兴趣对象开始的第一件事情，这是很容易找到的。给出指向任意一个这种对象的指针，任何函数就能找到它的vptr。例如，可以对类对象遍历vtbl的内容。其关键因素是在运行程序中动态地替换函数。

Top类也有一个在protected节中的缺省构造函数（无参的），这样可以防止类Top的对象被直接地创建。Top类有一个什么都不做的虚拟析构函数；它是虚拟的可以确保在运行时所调用的合适的导出类析构函数被调度。

类Top的operator new的说明被隐藏在protected域内，确保没有信封类从尾部分配。限定信封类说明（是局部和全局作用域的对象，或者另一个类的成员对象），使编译可自动地完成预期的清除任务。这种操作在信封导出类中被重写，所以它们的实例仍然能够从尾部进行分配。在信件层次结构中动态地分配和类的解除分配在它们信件类内执行，以确保当信件类对象不再使用时能够及时地收回所占的内存。

type成员函数在导出类中被重写，返回指向关联样本的指针；它用运行时的类来更新，其描述在本章后面加以介绍。

Top类完全依赖于编译程序。这里，大部分C++实现所产生的代码在格式描述上是变化的，但用户应当注意移植工作是按顺序进行的。

Thing类

Thing类是所有信件类的基类。因为信件类包含了大部分应用的智能，所以大部分动态对象的语义在Thing的公有接口中可以找到。在符号习惯用法中，即使一些内存功能性管理——通常在信封类中找到——都在Thing继承层次结构下的类中实现。

deref 和 ref 函数控制引用计数私有成员 refCountVal。它们都是虚拟的，这样，导出类就能够重写它们。但是，典型的应用可用非虚拟方式来定义它们，如内联（inline）成员函数。这些函数根据原则约定表示而存在。私有成员 updateCountVal 支持增量装入，它的使用在本章后面将加以描述。

cutover 函数把给定类的已有对象转换到反映相同类的不同版本的对象中。这种对已有对象数据的转换允许从原始格式转换到新类格式，把类的新版本引入到运行系统之中。通常，函数在导出类中根据使用需要而被重写；它的缺省语义恰好返回指向原始实例的指针。

虚拟析构函数是下一个仅有的处理，以确保当 operator delete 用于 Thing 的导出类对象时，相应的导出类可以被清除掉。

应用类的符号传统格式

在 k.h 中有类的框架，现在我们可以很容易地在符号习惯用法中刻画应用类所使用的规范格式的特性。我们采用两种基本类：信封类，用于检查对象的建立和分配；信件类，收集了大部分应用的系统语义。

一个给定信封类可以与几个信件类有关。让我们来设计并指定 Number 类型，此类型成为 Double，BigInteger 和 Complex 的基类型。常规的 C++方法使用继承和虚拟函数，通过 Number 接口允许这些类的对象是可互换的：Number 应该是其他 3 个类的抽象基类，并且 Number 是没有实例的。在符号习惯用法中，用户仍能通过 Number 接口处理所有数（number）类的可互换的对象。但是，Number 恰好作为信件类 Double，BigInteger 和 Complex 之一的信封。这些信件中的 3 个信件类都是从通用基类 NumericRep 中派生出来的，一个类的特征由它的导出类来刻画：它是一个数应用的“通用信件”。Number 信封包含了一个指向类型 NumericRep*说明的指针，它的使用“包含”着信件。通用信件类依次从 Thing 类导出，而 envelope（Number）是由 Top 类导出的。这种结构给强大的多态性和运行时更新添加了合适的间接层次。

注意，因为信件是由 Thing 导出的，而信封由 Top 导出，所以我们不能像以前看到的例子那样使用信封和信件类的公共基类。

为了概括符号规范格式，我们将使用类属的例子，这个例子具有 Envelope 类（是信封类）和表示通用信件基类的 Letter 类。这是一个合成的块，即由 Envelope 对象和从 Letter 导出的类中的对象组成，表示单一抽象，能够作为有符号习惯用法能力的、灵活的程序构造块使用。

任何程序或系统可以有许多信封类遵循 Envelope 的规范格式，并且许多信件类都模拟 Letter，每个信件类都定制了各自的语义。例如，一个程序可以有一个模拟 Envelope 的 Number 类，NumericRep 模拟 Letter 类，以及 Complex 类和其他从 NumericRep 中导出的类（正如上面描述过的）。相同的程序可以有 Shape 类，也可以模拟 Envelope 和 ShapeRep 以及模拟 Letter 层次结构的树中相关联的导出类。虽然 Number 和 Shape 对象的组合是独立的，但它们可以进驻同一程序，每种都用符号习惯用法构成。

例 9.2：Envelope 类。

```
#include "k.h"                          // 从上面而来
#include "Letter.h"                     // 为 Envelope 成员函数
extern Thing *envelope, *letter;        // 样本指针
class Envelope:public Top {             // 在 k.h 中定义 Top
public:
```

```
    Letter *operator ->() const {          // 对 rep 转发所有操作
        return rep;
    }
    Envelope()   { rep = letter ->make(); }
    Envelope(Letter&);
    ~Envelope()  {
        if (rep && rep ->deref() <= 0) delete rep;
    }
    Envelope(Envelope& x)  {
        (rep = x.rep) ->ref();
    }
    Envelope& operator = (Envelope& x)  {
        if (rep != x.rep)  {
            if (rep && rep ->deref() <= 0) delete rep;
            (rep = x.rep) ->ref();
        }
        return *this;
    }
    Thing *type()  { return envelope; }
private:
    static void *operator new(size_t)  {
        Sys_Error("heap Envelope");
    }
    static void operator delete(void* )  {}
    Letter *rep;
};
```

Envelope（见例 9.2）总的描述了一个处理实例化和分配的类（即，它处理所有的“拷贝”和许多内存分配管理工作）；Envelope 经常使此类成为具体数据类型，并赋予完美的传统规范格式（见 3.1 节），就此而言，这是很有用的。信封有点像标号，可以贴在不同的对象上。分配的意思是和对象的这样的标号相关联的，剥去最后一个标号的意思是返回该对象的自由存储区。

信封类的行为像一种松散类型抽象，而它的实例模拟变量，其动作在许多符号语言中就像无类型的对象标号。例如，信封的成员函数不传送包含在各种对象内的细节语义。但是，信封对象使用它的信件类对象的行为，这是因为使用了 3.5 节中描述的 operator->机制，就好像在符号语言中的变量使用了受到约束的对象的行为那样。这里有一点需要提示，即信封的接口传送一些它的信件类的消息，是在 operator->返回类型中，即在 letter*中。这是在符号习惯用法中，对应用程序员指针是可见的几个地方之一，但它仅是个过渡值，是非正规地获取的。

信封类有一些构造函数，但大多数显式对象初始化是用信件类的虚拟 make 函数完成的；以后将会展示，这是一个渐进性的辅助办法。信封构造函数是初始化的“特写”，并通过编译程序转换而执行。这两种构造函数均来自于传统的规范格式：缺省（无参）构造函数和拷贝构造函数。最后，需要被转换的构造函数都从信件类的实例中创建新的信封对象。这种构造函数转换了信件类对象内部计算的结果，可以被信封/信件聚合对象的一般用户所使用。

拷贝构造函数、分配操作和析构函数维护了对象的引用计数（如 3.5 节中所描述的）。析构函数检测它的信件的引用计数是否为零，或者如果最后的信封停止了引用，则收回信件所占用的空间。

最后，或许最重要的是 operator ->，即自动地转发信件对象对信封成员函数的调用。如果信封

复制了信件的特征，并且每个信封函数恰好具有控制它的信件事务转发的能力，那么就具有相同的作用。但是，这样会导致添加新的信件类成员函数的复制工作。

例 9.3：Letter 类。

```
class Letter:public Thing {
public:
    /* 这里是所有用户定义的操作符的运作。注意：因为使用了->操作符，
     * 这种特性不一定要在信封中模仿。但信封中的 rep 域必须是合适的
     * 类型。赋值操作符不在这里运作，而在信封中运作。
     * return_type 应当是一个基本数据类型，
     * 或者是 Envelope 类型，或者是 Envelope& 类型，
     * 或者是一个具体数据类型。
     * /
    virtual void send(String name, String address);
    virtual double postage();
    virtual return_type user-defined-function;
    ...
    virtual Envelope make();            // 构造函数
    virtual Envelope make(double);      // 另一个构造函数
    virtual Envelope make(int days, double weight);
    virtual Thing *cutover(Thing*);     // 实时更新函数
    Letter() {}
    ~Letter() {}
    Thing *type();
protected:
    friend class Envelope;
    double ounces;
    static void *operator new(size_t 1) {
        return ::operator new(1);
    }
    static void operator delete(void *p) {
        ::operator delete(p);
    }
    String name, address;
    ...
private:
    ...
};
```

例 9.4：Letter 类的内联函数。

```
/*
 * 把一般的内联放在这里，这是为了支持
 * 内联产生在类说明中。所以，在信封和信件
 * 类之后放置内联是为了中止一些循环依赖。
 * /
inline double
Letter::postage() {
    if (ounces < 2) return 29.0;
    else return 29.0 + ((ounces - 1) * 23.0);
}
```

```
inline Thing*
Letter::type() {
    extern Thing *letter;      // 样本
    return letter;
}
```

Letter 类（见例 9.3 和例 9.4）用所有为 Envelope 接口服务的类来定义接口。Letter 本身是一个基类，通常是一个抽象基类，对象的类组合是用 Envelope 来进行接口服务的。一个给定的 Envelope 对象可以在它的生命周期内为几个不同的信件对象的接口使用。例如，Number 对象可以初始化 Complex 信件对象的接口，但分配和其他的计算可以产生用 Number 替代另外一个对象的原始信件，很可能在不同的类中。

在 Letter 层次结构中的对象被认为是在 Envelope 内的对象，Envelope 对象仅对用户直接可见。Letter 的特性绝不能被用户直接访问。但是，Letter 通过 Envelope::operator->的工作对 Envelope 接口贡献了它的成员函数。Letter 不需要变为具体数据类型，因为类型为“C”结尾的类型在 Envelope 层上被处理了。

Letter 类是作为用 Envelope 相关应用类所管理的基类。Envelope 类包含了指向 Letter 实例的 rep 域，所有“真正”的工作是在 Letter 导出的类的对象中进行的。

所有应用的成员函数都在信件类的接口中被指定，通常作为纯虚拟函数。从信件类中导出的所有公共类的一些函数可以分解到信件类本身（作为因子），并在基信件类定义中含有类体（body），使得其余的纯虚拟函数确保它们在导出类中被定义。但是，在本章后面我们将会看到，纯虚拟函数不接受这种扩充的与样本对象特性混合的习惯用法格式。

用户定义的函数应当既可以返回内联类型的对象，又可以返回具体数据类型的对象（换句话说，其中之一是附着在传统的规范格式之上的），或者是 Envelope 类型，或者是 Envelope 类型的引用。只有出现在 Envelope 特征中的指针类型才是 const 指针；返回不受保护的指针，指向动态内存分配块，这样就削弱了内存管理机制的基础。当然，返回值的声明也可以是 void 类型。

make 函数构造了从 Letter 导出的类的实例，并返回 Letter*，就像第 8 章所介绍的那样。可以有好几个重载的 make 函数，每个函数必须要做的工作是初始化新的对象。做什么事情也不能脱离开初始化构造函数。例如，Letter::make 可以初始化 OverNight 类和 FirstClass 类的对象，类似于：

```
Envelope
Letter::make(int days, double weight) {
    Letter *retval;
    if (days < 2 && weight <= 12) {
        retval = new OverNight;
    } else {
        retval = new FirstClass;
    }
    retval ->ounces = weight;
    return Envelope(*retval);
}
```

如果该构造函数留给不具备特性的初始化逻辑来做的话，那么就绝不需要再编辑和重新编译：在增量装入环境中的那个初始化值，其中虚拟函数（诸如 make）可以渐进地装入，但构造函数却不能。

一般地，Letter 导出类无须遵循传统的规范格式。应该有缺省（无参）的构造函数和析构函数，

但无须拷贝构造函数和分配的规格说明。

Letter 导出类遵循下面的约定。如果 Envelope 合理地设计，而且 Envelope 可以“包含”这些类（Letter 的导出类）的任何一个，那么 Envelope 显而易见地可以被其他的导出类所指定。例 9.5 展示了一些简单的从 Letter 导出的类，每个这样的类也有一个缺省的构造函数。

例 9.5：独特应用的具体的 Letter 类。

```
class FirstClass : public Letter {
public:
    FirstClass();
    ~FirstClass();
    Envelope make();
    Envelope make(double weight);
    ...
};

class OverNight : public Letter {
public:
    OverNight();
    ~OverNight();
    Envelope make();
    Envelope make(double weight);
    double postage()  {return 8.00; }
    ...
};
```

下面的样本习惯用法，其中每个信封类都有一个单一的作为样本的可访问的对象：它可以用专门的构造函数来创建，采用这个构造函数的对象与“原始”应用的对象之间具有一定的差异。通常对所提供的信件类的样本是有用的，所以信封样本有一个引用信件的实例。信件样本域对象创建的要求是，所有 make 调用由信封的 operator->方法转发到信件对象。信件样本可以是通用的基信件类（这里是 Letter）的特殊实例，如果那个类不是虚拟基类。如果是虚拟基类，那么就能构造出特殊的信件导出类，此类具有缺省的纯虚拟函数，以此来创建惟一的信件样本对象。

符号规范格式的类对计数指针和样本对象来说，都采用相同的方式来使用，即，使用“->”来替代“.”操作。下面有一个简单的应用，列举了我们在教学过程中所使用的 Envelope 和 Letter 类：

```
static Envelope envelopeExemplar;    // 从未直接使用
Envelope *envelope = &envelopeExemplar;


int main()  {
    Envelope overnighter = (*envelope) ->make(1, 3.0);
    overnighter ->send("Addison-Wesley", "Reading, MA");
    Envelope acrosstown = (*envelope) ->make(1.0);
    overnighter = acrosstown;
    acrosstown ->send("Angwantibo", "Boston Common");
    return 0;
}
```

这些都是符号规范格式的基础。为了强调和扩充以上所介绍过的内容，下面，我们归纳了该习

惯用法的一些规则：

（1）所有对 Envelope 类的引用应当通过“->”操作符，而不是用“.”操作符。这就允许 operator->自动地转发此操作到 Letter 类。

（2）信件类的成员函数应该是虚拟的。虚拟成员函数正如本节所描述的那样，很容易渐进地装入。

（3）make 成员函数履行构造函数的工作，构造函数本身则只做很少的工作。这是因为我们要求在运行时能够替代类的初始化代码，并且只有虚拟函数在运行时才能更新。一些构造函数仍在 Thing 导出的类中出现（它们必须是用 C++虚拟函数建立的），但它们不应当包含在用户提供的代码中。

（4）每个类应当有一个简单的永久样本出现在程序中。样本对象必须能够标识自己（例如，用已经建立的特殊的构造函数）。此样本绝不能直接访问，而只能通过一个指定的指针变量来访问。

（5）在 Thing 的子类中的 cutover(Thing*)成员函数用于高级运行时动态地装入。这些函数传送一个指向属于该类的对象的指针，它们的工作是把在原始的旧类转换到新类中的相适应的格式、布局和类型变更。如果 cutover 不能转换原始的对象，那么它可以在预先的环境基础上获得一些技巧，以使对象总能做些转换工作（即，模仿了 Smalltalk 的单向 BECOMES 的机制）。

（6）注意，new 绝不能够在信封类上执行。该操作是 private 的，倘若我们试图动态地分配信封就会导致编译出现错误。信封应该只说明为自动变量，其他类的成员可以作为全局变量。当回收对象存储区时，消去了指针就等于免除了记忆的沉重的负担，这就意味存在着像 Number 那样的灵活性、多态性类型。当然，也可以当作具体数据类型来使用，这就好像它们是 int 一样。

概括这种习惯用法可以帮助软件工程师进行有效的开发：内存管理基本上是用信封类自动地进行，因为当 Thing 变成不再引用时，它始终处于监视状态。当函数和数据的访问服从于附加的间接层时，这个信息的用户对类的改变具有一定的影响，这就意味着当类改变时连锁反应少，并且 Thing 依赖于需要重新编译的类的实现细节。给出合适的连接编辑和装入工具，这种技术就能支持运行程序增量地改变类，在新类中也就不需要那么多的程序配置。

这种方法增加了对 C++语言多态性的力度，提供了运行时参数化类型和类属类机制的分类。

9.3 通用收集类的例子

例如，考虑一个程序，我们希望使用 3 种不同的可互换的种类：一个是整数下标的数组；一个是 B 树的数组，还有一个是杂凑表（hash table）的数组。为此，我们定义一个信封类 Collection，它包含指向以上内部对象的指针。为了增加灵活性，将 Collection 作成模板，使之能够实例化所希望包含的类对象。这样的说明为

```
Collection < Book, Author > library;
```

创建一个我们希望的用作者索引 Book 对象的聚合。

这个例子的信件对象由 CollectionRep 导出的类来创建。这些类描述了 CollectionRep 变体的特征。这里，3 个 CollectionRep 的变体（Array，Btree 和 HashTable）替换了 Collection 类，可以作为一个包容器对象而被使用。CollectionRep 的大多数成员函数及其导出类应当是虚拟的，所以从 Collection 调用的成员函数通过 CollectionRep 指针发送到合适的 Array，Btree 或 HashTable 成员函数中去。CollectionRep 类接口必须说明所有这些类的成员函数联合的操作。只有进行了信件类层次

结构的操作，才知道 Collection 的这些说明是在 CollectionRep 它自身。因为不是所有的信件都重写所有这些成员函数，所以例外（异常）处理需要从不合适的成员函数引用中收回所占据的存储空间。例如，Array 只能用整数索引进行访问，HashTable 可以使用整数索引进行查找，也可以用相关的字符串进行查找。如果当前 Collection 对象保存 Array，那么 operator[ ](s)的引用就会产生例外。这是我们为符号程序设计风格的灵活性所付出的代价。

例 9.6：Collection 表示信件的基类。

```
#include "k.h"
#include "Collection.h"
// 存储 T类的项，用 S类的值索引
template < class T, class S >
   class CollectionRep: public Thing {
   public:
      virtual Collection < T, S > make();
      virtual Thing* cutover(Thing*);
      virtual T& operator[](int);
      virtual T& operator[](S);
      virtual void put(const T&);
      CollectionRep()  {}
      ~CollectionRep()  {}
      Thing *type();
protected:
   friend class Collection < T, S >;
   static void *operator new(size_t 1)  {
      return ::operator new(1);
   }
   static void operator delete(void *p)  {
      ::operator delete(p);
   }
private:
   CollectionRep < T, S > *exemplarPointer;
};
```

例 9.6 表示了 CollectionRep 说明的框架，是这个例子的信件类的基类。在 CollectionRep 层次结构中的类使用独特的机制来实现 type 成员函数。因为 CollectionRep 是参数化类（到新类中的每个实例要求它自己（潜在的人工管理）的样本），所以 type 成员函数不能返回全局指针值。样本进行 make 操作，使得在每个对象内储存样本地址（在 examplarPointer 域中），并且 type 成员函数返回其值。例如，

```
class ClassDerivedFromCollectionRep : public CollectionRep  {
public:
   ...
   Collection < T, S > make()  {
      Collection newObject;
      ...
      newObject.exemplarPointer = this;
      return newObject;
   }
```

```
    ...
};
Thing *CollectionRep::type() {return exemplarPointer; }
```

CollectionRep 类是有用的抽象，并且能够直接使用另一个习惯用法。在 Collection 内，逻辑上封装 CollectionRep 层次结构有两点好处。首先，它允许 Collection 在运行时要求改变它的表示。动态输入使之可能把一种类型收集分配到另外一种类型的收集之中，所以，所有的收集类型几乎都能够互换使用。大型的收集操作可以使用尺寸界限值或其他准则，在运行时改变它们自己的类型，以改善性能（例如，把线性 Array 转换成线程 HashTable）。第二，Collection 基类 Top 使用信封类的基类 Thing 的属性，实现样本习惯用法的透明内存管理。

CollectionRep 类框架图解了例 9.6 中的这种概念，从它的基类 Thing 中继承了引用计数机制。

例 9.7：收集类（Collection class）。

```
#include "k.h"
template < class T, class S > class CollectionRep;
template < class T, class S >
    class Collection:public Top {
    public:
        CollectionRep < T, S > *operator ->() const { return rep; }
        Collection();
        Collection(CollectionRep < T, S > &);
        ~Collection();
        Collection(Collection < T, S > &);
        Collection& operator = (Collection < T, S > &);
        T& operator[](int i) {return (*rep)[i]; }
        T& operator[](S s) {return (*rep)[s]; }
    private:
        static void *operator new(size_t) { return 0; }
        static void operator delete(void *p) {
            ::operator delete(p);
        }
        CollectionRep < T, S > *rep;
    };
```

例 9.7 展示了 Collection 类本身。它执行基本的内存管理（主要地，处理分配），换言之，把它的操作转发到它的信件类中。不管采用什么样的选择标准，它都能够从其所选择的类中创建（或重新分配）信件。对 Collection 添加构造函数可为用户提供控制所使用的数据结构。

例 9.8：某些 Collection 实现的类。

```
template < class T, class S >
    class Array:public CollectionRep <T, S> {
    public:
        Array();
        Array(Array <T, S>&);
        ~Array();
        class Collection <T, S> make();
        class Collection <T, S> make(int size);
        T& operator[] (int i);
        void put(const T&);
```

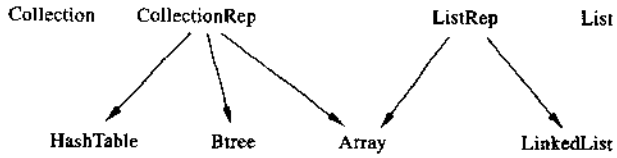
```
    private:
        T *vec;
        int size;
    };

template<class T>
    struct HashTableElement {
        HashTableElement *next;
        T *element;
    };
template<class T, class S>
    class HashTable:public CollectionRep <T, S> {
    public:
        HashTable();
        HashTable(HashTable<T, S> &);
        ~HashTable();
        class Collection<T, S> make();
        class Collection<T, S> make(int);
        T& operator[](int i);
        T& operator[](S);
        void put(const T&);
    private:
        int nbuckets;
        virtual int hash(int 1);
        HashTableElement <T> *buckets;
    };
```

CollectionRep 的子类如例 9.8 所示。每个 Collection 建立适合 Array，Btree 或 HashTable 的类作为信件。每个导出类重写了这些成员函数，使其表示是有一定的意义，剩下的就是在 CollectionRep 中的缺省的例外处理。注意：因为导出类选择性地重写了仅为基类函数的子集，所以这些函数不能在 CollectionRep 中用纯虚拟来说明。

通常使用引用计数来管理这种对象的存储区。在 3.5 节中所描述的类是信件/信封习惯用法应用的较好的例子，它采用引用计数。3.5 节展示了如何分配、构造成员函数和析构成员函数使之能够操纵引用计数的情形，也就是说包含了 StringRep 对象的成员，当计数为零时，就应删除这个对象。这里，CollectionRep 与 StringRep 是相似的。在 9.5 节中，我们将研究垃圾收集（garbage collection）的引用计数的替换办法。

这种方法可以有效地使用递增式的多态性。例如，采用具有任意内部数据结构的两个收集的联合。但是，这样做也存在许多需要考虑的隐蔽的毛病。尤其是，如果采用这种方式的类具有双目操作（诸如，通过 operator+合并两个收集），那么它们必须适应两个对象可能传递不同的实数类型（例如，合并 Array 和 Btree）。对这样的类的设计变得十分复杂，所以在考虑运行时的开销时就必须指出提供什么种类的对象和引用正确的操作（相关的模拟的例子，可参看 5.5 节中 Number 的例子以及下面 9.7 节中的讨论）。在理论上我们能在 Top 类层或 Thing 类层上定义这些转换，以提供完备的多态性。但是，这些类的定义是很复杂的，并且当添加任何新对象操作时，就需要改变。多重继承能够定义多个外部接口类（例如，是 Array 或 LinkedList 的 List 类共享了具有 Collection 类的 Array 类）例 9.9 展示了这样一个例子的类层次结构，然而很快就会陷入到复杂境地。

例 9.9：多重继承的类层次结构。



9.4 支持增量装入的代码和习惯用法

如果系统的增量连接编辑是有效的，那么通常允许编写装入程序，使得运行程序能够添加新的代码。例 9.10 展示了简单的 load 函数，此函数把重定位编译对象文件名作为参数，装入然后运行。假设对象文件 incr.o 只包含一个单一的函数，并且它的入口点是在对象文件的文本节处。这样我们就能够装入并执行这个函数。在运行时具有以下引用形式：

```
int main() {
    typedef void(*PF) (...);        // 指向函数的指针
    PF anewfunc = (PF) load("incr.o");
    (*anewfunc) ();
    return 0;
}
```

该程序在大多数 Sun Microsystems 平台上运行，程序本身必须使用-n 标记装入。类似的代码也可在大多数现在流行的操作系统平台上编写。

例 9.10：在 Sun 平台上装入单一函数的.o 文件的函数。

```
#include <a.out.h>
#include <fcntl.h>
#include <sys/types.h>
caddr_t load(const char *filename) {
    caddr_t oadx = (caddr_t)sbrk(0);
    caddr_t adx = ((char*)oadx) + PAGSIZ -
        (((char* )oadx) % PAGSIZ);
    sprintf(buf, "ld -N -Ttext % X -A a.out % s -o a.out.new",
        adx, filename);
    system(buf);
    int fd = open(filename, O_RDONLY);
    exec Exec;
    read(fd, (char* )&Exec, sizeof(exec));
    sbrk(PAGSIZ - (oadx % .PAGSIZ));
    caddr_t ldadx = (caddr_t)sbrk(Exec.a_text +
        Exec.a_data + Exec.a_bss);
    read(fd, ldadx, Exec.a_text + Exec.a_data);
    close(fd);
    return ldadx;
}
```

我们能把这样装入的程序的代码放到交互的 C++程序内，并安排在运行时装入新的函数。当程

序处于等待状态时，我们能够编译新代码，然后请求该程序装入到新的重定位的代码之中。添加的工作必须由联编函数引用已有的代码，然后就可以继续执行。

当使用这种习惯用法时：这里，习惯用法所构成的描述代码能够在开发环境中用人工或半自动的方式产生，以支持增量式开发。此风格能在运行程序中重要的动态变化的地方使用，通常是在原型中使用。这种习惯用法对大型支撑的领域以及需要连续运行软件的应用也很有用。

下面我们来介绍代码动态装入到运行程序中的习惯用法和结构。这个问题还可以分解成两个子问题：首先描述装入新函数的能力；其次描述转换已有对象的数据格式的能力。

装入虚拟函数

装入程序的直接应用就是把函数的新版本装入到运行程序中。如上所述，如果我们所生成的对象文件不是别的，只是装入一个新的函数而已，那么装入一个函数是件很容易的事。有些添加的工作是需要把新使用的函数联编到已有的函数引用中，并在此加以描述。

9.2 节描述了指向虚拟函数的指针表与每个表的关联。而且，对任何类的样本对象都用它的第一个元素作为指向那个表（它的 vptr）的指针，以确保所采用的符号格式是规范的（不同的编译程序其实现也是不尽相同的）。这就是为什么所有信封类都是由 Top 导出的缘故：它包含了对象的虚拟函数指针，其中我们习惯使用 vtbl 的地址[1]。

管理的手段必须在 C++环境中确保对每个类（除了从其他类的多重继承之外）的程序包含一个虚拟函数表的一个拷贝。这对于某些 C++编译系统来说是直截了当的，并且可以要求一些在其他编译系统重用的工具。这要检查一下编译程序所提供的帮助和在这个领域的指南。

仅对虚拟函数描述字表处理是不够的：我们需要知道该表的哪一个元素对应于我们所期望更新的函数。如果内存中的表存在着映射到 vtbl 索引上的函数的名字，那么就会产生意外。但是，在大多数 C++实现中这是不会发生的，所以有些工作必须从程序外部做起，以便给定装入的信息是所需要的、标识合适的 vtbl 槽。

做这项工作的一种方式是编写一个小的**装入辅助函数**，它要做的工作只是返回刻画该函数的聚合值，包括它的地址和它的虚拟函数表的索引。这个函数准备对函数的更新采用人工编码或自动生成方式，更新的过程分两个阶段。第一个阶段，辅助函数由装入程序装入，并产生所期望的函数的 vtbl 索引；第二个阶段，该函数的新拷贝装入并把它的地址连接到相适应的 vtbl 槽中。

在查找装入函数之前，我们需要首先查找某些支持的数据结构。首先，我们需要一个 typedef，它指向函数的指针：

```
typedef int (*vptp) ();
```

mptr 结构说明了虚拟函数表入口的状态，很像大多数已有的 C++实现：

```
struct mptr {
    short d;
    short i;
    vptp f;
};
```

这个说明基于 cfront 的 C++编译程序所使用的 C++系统，并且是许多系统的典型的使用。其他的编

1 此工作仅对单继承而言；多重继承一般是很复杂的，此处不包括。

译程序可以使用不同的结构，而且本章所描述的代码必须经过适当的修整。两个 short 习惯为多重继承的偏移，在此不再进行讨论。令我们最感兴趣的是 f 成员，它指向给定表入口的函数。

装入辅助函数很容易编写：所有需要做的工作是返回取代这个函数已存拷贝的地址。让我们来调用 functionAddress，即：

```
extern vptp functionAddress() {
    // 此代码依赖于机器和编译程序
    return (vptp) &Array::put;
}
```

每次装入一个函数，首先装入 functionAddress 的新拷贝，任何一个拷贝重写前一个版本——返回由正文内存管理的正文空间，或者如果内存的回收不是很严重的话，就不再回收未收集的垃圾。

当装入一个有重载名（使用合适的选型）的函数时，functionAddress 函数就能够判定二义性。例如，假设 Array 类有多个 put 函数：

```
class Array {
public:
    ...
    void  put(int, double);
    void  put(int, int);
};
```

左边指定合适的类型就能够使函数选择的第二个参数为 double：

```
extern vptp functionAddress() {
    // 此代码依赖于机器和编译程序
    typedef void ((Array::*TYPE) (int, double));
    TYPE retval = &Array::put;
    return (vptp)retval;
}
```

函数的下一个组合是添加 Top 类，以支持增量装入和函数替换。首先，CompareFuncs 计算两个函数的描述是否是指定的同一函数：

```
int
Top::compareFuncs(int vtblindex, vptp vtblFptr, vptp fptr) {
    // 此代码依赖于机器和编译程序
    return vtblindex == (int)fptr;
}
```

一对自变量中的第一个传送虚拟函数表入口的信息。int 是它的索引，并且 vptp 自变量是在入口处找到的函数指针（mptr ::f 成员）的内容。第二个参数是所替换的函数的地址。如果（从两个参数中的第一个）CompareFuncs 能够确定该函数的表示与最后一个自变量的表示相同，则返回非零值。实际上，无论哪个参数采用这种方式加以比较都依赖于实现。这里，我们知道，当使用虚拟函数的地址时，编译程序就会产生一个虚拟函数的虚拟索引，所以，需要所有的参数与索引参数进行比较。

第二个函数是 findVtblEntry：

```
mptr *
Top::findVtblEntry(vptp functionAddress) {
    // 此代码依赖于机器和编译程序
```

```
    mptr **mpp = (mptr **) this;
    register mptr *vtbl = *mpp;
    for(int i = 1; vtbl[i].f; ++i ) {
        if (compareFuncs(i, vtbl[i].f, functionAddress) {
            return vtbl + i;
        }
    }
    return 0;
}
```

这个函数在当前对象的虚拟函数表中（指向对象中的第一个字 vptr）查找相对应的被更新的函数的入口，用此作为参数来传送。如果能查找到，则此函数返回一个指向相应的虚拟函数表入口（mptr）的指针。

这样，剩下来的事就是装入和联编新的虚拟函数。这是用 Top::update 来加以协调的：

```
extern "C" vptp load(const char *);
void
Top::update(const char *prepname, const char *loadname) {
    vptp findfunc = load(prepname);
    mptr *vtbl = findVtblEntry((*findfunc)());
    vtbl ->f = load(loadname);
}
```

它调用了包括装入辅助函数的文件名以及包括被装入函数的文件名。使用以前定义的函数（用 load 返回 vptp 重写类型替代 caddr_t）找到虚拟函数表入口，并替换函数指针域，这个指针域具有指向重新装入函数的指针。对于这个虚拟函数的所有下面的引用都与新版本捆绑在一起。

还有一点要说明的是，该代码能够调节类的新虚拟函数的添加。但是，对这样的机制的完全的支持要求有更高级的管理工具的配置，以便既能维护语义的正确性，又能保持语义的渐进性。例如，添加一个新的虚拟函数需要的不只是刚生成的 vtbl，而且还要确保以前保存的实体的次序。考虑添加一个新的虚拟函数，其中名字隐藏了以前存在的全局的函数：如何评估所得到的重编译和重装入的工作？

类布局更新和 cutover 函数

为了更好地拓宽处理数据布局的变化，增量更新技术是一种很好的方法，可极大地扩充开发的灵活性和产品支持环境。这个问题比函数更新更困难：虚拟函数提供一个在数据中未建立的间接层。但是，信件/信封的习惯用法恰好具有我们正在寻找的数据间接附加层次，并且它能够平稳地执行数据的改变。本节将描述在信件类中所支持的增量数据改变的一种方法。因为大部分的应用代码驻留在信件类中，所以，信件类更新技术就可以处理大部分成员函数的改变。

更新类数据布局或重新装入一个类的意思是重新装入它的成员函数；这里不存在相对应的类布局本身的代码，但该布局的信息通过它所操作的正文来分发。前一节我们描述了如何做这样的增量函数的装入，但重新装入一个类不仅要重新装入所有的成员函数，还必须把现存的对象转换到新类的布局之中。

为了支持这种能力，每个样本始终监视着每个请求建立的对象。按此种方式，当它们的类被改变时，就能够转换每个信件类的数据表示。一个样本能够用单一的方法很方便地监视它的实例，一般目的的 List 类从 C++库中获得。List 对象可以在样本类中说明的静态成员。

用户必须提供一个 cutover（转换）成员函数，此函数知道如何把已有的内存对象转换成新版本，以改变它们的格式但保留其语义。这个函数就像任何其他函数一样，能够独立地装入。装入后所引用的 cutover 函数完全能够胜任应用本身的基础结构。当该系统处于熟知的静止状态时，此应用就可以挑选所引用的 cutover——例如，就在此时我们知道激活的记录是不可能打开任何正在更新的记录的。

许多方法可以采用 cutover 函数这种手段，下面的例子给出了需要做的这些事情的范畴。

在类中添加一个新的域

让我们看一下要在 HashTable 类中添加的新域，一个溢出存储段列表为

```
template < class T, class S >
   class HashTable:public CollectionRep < T, S > {
   public:
      HashTable();
      HashTable(HashTable < T, S > &);
   ...
   Thing *cutover();
private:
   int nbuckets;
   virtual int hash(int 1);
   HashTableElement < T > *buckets;
   HashTableElement < T > *overflow;      // 新的
};
```

注意，新的数据项已经添加到这个类的末尾。通常这么做有一些优点，即对旧的接口编译的代码仍可以使用从新接口建立的对象，也可以使用简单的 cutover 算法。值得注意的是，在新类中 cutover 函数已经被重写。

如果原始元素的数据布局在添加新的数据项中未受到影响，那么 cutover 函数就很容易设计：

```
Thing *
HashTable::cutover() {
   HashTable *object = (HashTable*) this,
             *retval = new HashTable;
   // 拷贝基类--部分
   ( *(ColectionRep *)retval) = (*(CollectionRep *)this);
   // 拷贝这个类的域
   retval ->buckets = object ->buckets;
   // 初始化新的域
   retval ->overflow = 0;
   // 删除旧的域
   object ->buckets = 0;
   delete object;
   // 返回新对象
   return retval;
}
```

这里，虽然改变是由 HashTable 模板本身造成的。该模板需要重新实例化在程序内使用的每个参数，即每个 HashTable 的实例化需要分别加以更新，尽管源代码的转换只须进行一次。

在类表示中取代改变

如果一个类的内部固有特性被改写、取代的话，那么更先进的手段就必须是更新已有的实例。新实例需要由旧实例中推导出来。cutover 函数需要同时访问该接口的旧说明和新说明。我们要拷贝旧类接口的说明，此说明把所有可能出现的类名变换到某些暂时（类）名中。这个接口和新接口同时都使用 cutover 函数，所以 cutover 不受新对象如何从旧对象建立的限制。

作为一个例子，我们考虑 Point 类：

```
class Point : public ShapeRep {
public:
   Shape make(double x, double y);
   void rotate(Shape& p);     // 旋转一个点
private:
   double x, y;
};
```

我们希望利用某些新的图形硬件使 Point 旋转一个角度（弧度）。这项工作并不改变它的接口，所以我们可以转换已有的对象，而系统仍能正常工作。新类可以这样设计：

```
class Point:public ShapeRep  {
public:
   Shape make(double r, double theta);
   void rotate(Shape& p);     // 旋转一个点
   Thing *cutover();
private:
   double radius;
   Angle theta;               // 由 double 很容易建立
};
```

我们根据暂时类名来改写旧的头文件，其结果产生以下接口。注意，Point 说明为 friend，其目的使 cutover 能够直接访问 Point 的域。在大部分情况下，cutover 应该能够从旧类的公共特性中获得所需要的东西。但是，在更新之后，旧函数就再也不能使用了！

```
class OLDPoint:public ShapeRep {
friend Point
public:
   Shape make(double x, double y);
   void rotate(Shape&  p);     // 旋转一个点
private:
   double x, y;
};
```

下一步，我们来编写新类的 cutover 函数：

```
Thing *
Point::cutover() {
   OLDPoint *old = (OLDPoint *)this;
   Point *newPoint = new Point;
   newPoint ->radius = ::sqrt(old ->x *old ->x + old ->y *old ->y);
   if (::abs(old ->x) < .000001)  {
      newPoint ->theta = ::atan(1) * 2;
   } else {
```

```
        newPoint ->theta = ::atan2(old ->y, old ->x);
    }
    if (y < 0) newPoint ->theta += ::atan(1) * 2;
    return newPoint;
}
```

调节函数装入和对象转换

应用本身必须调节装入和转换活动。例如，需要知道在执行更新时是否安全（应避免在关键性的计算当中更新），或可能需要得到用户的输入（得到包括新对象代码的文件名字等等）。这样，“调节的代码”就能自动地产生许多应用程序。本节讨论如何对这样的代码进行设计的一些考虑。在此，我们还是用 Point 作为例子，让我们来浏览一下这种更新。

在 Point 源代码转换完成的情况下，所有 Point 的成员函数需要对新接口重新编译。由每个 Point 虚拟成员函数编写装入的辅助函数，然后已有的 Point 样本引用 Top::update 成员函数，把新的函数装到内存中。每次装入一个，与辅助函数组成的每个对相应地调用 Top::update。

现在，所有的函数已经装入。在新的函数当中是 cutover 函数把现有的实例转换到新的表示之中。所有的样本始终监视着所有驻留的实例，所以从信件中导出的类转换现有对象的工作是直截了当的。记住，信件类是在相应的信封类内的概念作用域中，所以我们知道，只有一个信封对象的类才能够用 rep 指针引用信件层次结构中的类的任何给定对象。这个信封类的样本可以跟踪它的所有的实例，而且依次调查每个信件是属于什么类型的。任何这样的 type 成员函数的信件都返回一个指向被更新的信件样本的指针，这些都是更新的候选者。这就意味着它能够在每个对象依次执行 cutover 操作，更新每个 rep 域，所以它指向转换的实例。

例如，一般认为正在取代的 Point 是从 ShapeRep 中导出的。Shape 的样本有一个由所有 Shape 对象构成的记录。它知道在 ShapeRep 层次结构中每个 rep 域中的一个指向某一个对象。搜索每个 Shape 的实例 *s*，更新算法集中在 s.rep->type() == point 是“真”的这方面，其中 point 指向 Point 样本。对每个这样的 *s*，用 s.rep->cutover 取代 s.rep 的值。现在，所有信封对象都将更新新类的引用对象，从旧的对象中转换过来。

更加精确地进行调节是很必要的。如果多个信封共享同一个信件，那么就会把旧的和新的版本混淆，而且转换也就失灵了。我们需要更新每个共享就一次的信件。为了完成这项工作，我们在每个 Thing 对象中设定一个计数器，使之能够隐蔽引用计数。当访问每个对象时，由隐蔽的计数器检测是否为零（初始化值为零）：如果是零，那么计数器设定引用计数的值，然后隐蔽值递减；如果是零的话，那么就执行 cutover；否则，就跳过（执行空指令）。所以，每个共享的信件对象在访问更新扫描的最后一次被转换。

大多数这种调节可以放在新的 Thing 成员函数中，命名为 docutover：

```
int
Thing::docutover() {
    if (!updateCountVal) {
        updateCountVal = refCountVal;
    }
    return ! -- updateCountVal;
}
```

例 9.11：控制整个类更新周期的例子代码。

```
typedef Thing *Thingp;
void
Shape::dataUpdate(Thingp &oldExemplar,
                  const Thingp newExemplar) {
   Thing *saveRep;
   Shape *sp;
   for (Listiter <Shape* > p = allShapes;
      p.next(sp); p ++ ) {
      if (sp ->rep ->type() == &oldExemplar) {
         if (p ->rep ->docutover()) {
               saveRep = sp ->rep;
               sp ->rep = (ShapeRep*)sp ->rep ->cutover();
               delete saveRep;
         }
      }
   }
   saveRep = oldExemplar;
   oldExemplar = newExemplar;
   delete saveRep;
}
```

例 9.11 的 Shape::dataUpdate 函数，对从 ShapeRep（信件基类）中导出的类中所改变的数据，调节了更新。它的参数是信件类更新的样本指针，并且指向一个取代的样本的指针。在执行了上面描述的算法之后，它也就更新了该样本的对象，把完全的程序改变留给反映该类最新的装入版本。dataUpdate 函数假定该类的新版本的虚拟函数以前已经装入了。

增量装入和自主的类属构造函数

当使用自主的类属构造函数时（在 8.3 节中描述过的），增量装入是一种有效的工具。自主类属构造函数允许特化样本（如 Number，Name，Punctuation），注册在它们自己的具有更抽象的一般样本（Atom）中，我们称之为**自主类属样**本。通常，该注册是从自主类属样本的类中导出。自主类属样本作为它的所有注册的代理：注册的集合可以在程序生命周期中加以改变。

在增量装入的情况下，我们在程序生命周期中可以添加新的导出类，创建它们的样本，并用抽象基样本加以注册，所以对于这样的例子，我们可以将 BinaryOp 类和 UnaryOp 类添加到运行系统的语法分析程序中，以便使用合适的构造函数创建样本对象，并用 Atom 类的新样本注册。然后，在新串中从它的用户接口中系统地接受并分析双目和单目操作符表达式，接下来进行数据连接，并且不管在什么地方 Atom 样本总要使用句法分析。

9.5 垃圾收集

符号程序设计环境流行的特征是能否把程序员从繁重的内存管理工作中解脱出来：未引用的对象可以自动地在运行环境中回收（通常在操作系统内部，在硬件的帮助下，或两者都支撑）。这就是所谓的**垃圾收集**。垃圾收集给人一种似乎有无限大存储空间的感觉，所以应用程序员可以省心了，不需要考虑对象所需的空间以及什么时候不再需要这些空间。如果系统能够确定一个对象在任何地方不再使用的话，那么它所占的空间可以被回收，并且该空间可以分配给最需要的对象。对象的回

收机制及其安排对用户是透明的。

在 3.5 节中介绍的引用计数习惯用法是垃圾收集的弱格式。特别地，计数指针的习惯用法提供了符号环境中所要寻找的内存管理透明度的层次。但是，引用计数回收算法不能回收循环引用的数据结构，除非使用了昂贵的递归扫描技术。垃圾收集模式用于避免此种限制的高层面向对象程序设计环境，因此很少使用引用计数。本节介绍垃圾收集的习惯用法中的挑选引用计数技术。

非引用计数技术对实时嵌入系统也有好处，其中诸如内存或处理程序失败等例外事件可以使应用恢复层的内存资源不易回收。如果处理脱离实际并且需要终止的话，那么就不可能有机会执行它的析构函数。如果此处理与其他处理过程共享引用计数对象，那么这些对象的引用计数绝不会为零，并且该对象也不会被回收。内存资源审计的清除动作根据这些故障更常见的是富于弹性，根据这种回收能力，可以使引用计数更好地把无用资源回收到自由空闲池中。

大多数符号程序设计环境在编译和运行支持的环境实现中隐藏了垃圾收集细节，虽然某些早期的 Smalltalk 环境使用了引用计数内存回收算法，这些算法由应用程序编写非引用计数的代码。这与 3.5 节所描述的 C++方法形成了鲜明的对照，即用内存管理作为语言内的习惯用法实现，以替代编译和环境内隐蔽的算法。一些垃圾收集模式依赖于专用的硬件，以告知内存中的代码是否激活，并指向一个对象的指针，或者指向一片数据的指针。无论是由编译支持、操作系统机制支持，还是由专用硬件支持，符号语言内存管理都掌握在由应用程序员编写的源代码层之下。当使用“垃圾收集”术语时，通常这种透明性被隐藏。由于 C++语言非常贴近于机器，所以就不存在隶属于调节完全透明性的层次和通用垃圾收集机制。因此，我们采用支持简单方便提供的代码来营造垃圾收集这样一种环境，于是 C++程序就能够提供所挑选的类的主要的透明的垃圾收集。

这里，我们所描述的技术与 3.5 节的引用计数方法是等价透明的，并具有处理循环引用结构的附加值。但它的实时性能要比引用计数稍差一些。该技术的变型支持增量垃圾收集，所以正规地处理不一定要长时间地挂起以回收内存。而且，与便捷的垃圾收集器不同，它允许程序员与对象回收的一些活动有关：即垃圾收集器可以引用析构函数来消除要回收的对象。

已经有好几种算法在开发垃圾收集。一种早期的方法是“标记和扫描”法，即分析内存中的所有对象，看看是否被其他对象所引用，并且在被引用的对象上作出标记。在所有对象均被处理之后，就扫描并抛弃未打上标记的对象。

半空间拷贝是一种确保未被引用的对象完全被清除掉的一种技术。半空间拷贝的 Baker 算法[2]的原型实现，避免了用标记和扫描方法所发生的延迟现象。在 Baker 算法中，把内存分成两半，一半为 *A*；另一半是 *B*。*A* 空间通常称为“到空间”，面 *B* 空间称为“从空间”。新对象在 *A* 中创建。随着时间的推移，在某些点上（诸如当 A 被填满或系统空闲时），所有在 *A* 空间中达到的对象被压缩到 B 空间的末尾。指针从一个对象指向另一个对象，以调整映射的重新定位的位置。此时，*A* 就包含了未引用的对象：它是所有的“垃圾”。*A* 和 *B* 的角色是相反的，并且对另一个周期的处理过程是连续的。

Baker 算法（和大多数非引用计数垃圾技术）依赖于能够在数据值和对象引用（指针）之间加以区分。它不可能寻找到 C++内存中的任意代码，以确定它是作为指针使用还是作为整数使用。最近所发布的垃圾收集技术几乎都在使用基于标记和扫描的技术。这些垃圾收集器，可能会在回收内

2 H.G.Baker, “List Processing in Real Time on a Serial Computer”, A. I. Working Paper 139, MIT-AI Lab, Baston (April 1977).

存上造成失误，因为指针混淆了，内存中数据值出现的内容与对象地址同时发生，显然，该值表示为整数量值，或者表示为与对象地址解释不相关的一些事情。但是，这些算法只要程序员遵循合理的约定就不会遗留下悬挂的引用。

但是，在符号信封和信件的限定领域中，我们可标识给定信封类的所有对象（采用在它的样本中查找该表），所以我们能够找到对任意给定信件类的所有引用。如果我们对使用的固定的池（见 3.6 节）改变信件类的内存分配算法，那么在该池中所有的对象类型就是已知的（它们都具有相同类型）。我们也知道每个可能从信件类对象中找到的地址，所以希望按这些标记来使用和清除留下的空间。因为在类池中的这些对象都具有相同的大小，所以我们不用担心内存的破碎；折衷是每个垃圾收集类的对象所使用的大量的内存空间必须预先建立的策略。

当使用这种习惯用法时：使用垃圾收集能够把用户从繁重的内存管理中解脱出来。例如，可以支持快速原型环境。垃圾收集还是审核系统是否具有实时例外的强有力的技术。它可以确保所有内存资源的回收，即使是在例外情况下。

以 Triangle 作为例子，让我们来看一看符号信封/信件习惯用法的垃圾收集方法。为了支持这种技术，信件基类 ShapeRep 必须增加“标记”位，使之能够检查该对象是否“引用”（像算法执行那样）。这些对象也需要“使用中”位，以标明是否该空间是空闲的；这个符号位也能用 ShapeRep::operator new 置入使用，以便发现未使用的空间，满足对象建立的需求。在初始化时，所有“使用中”和“标记”位都加以复位；当一个对象被分配时，“使用中”位也就被设定了。此外，对这些位来说，每个对象有 *A* 位和 *B* 位，相对于对象是在 *A* 空间中还是在 Baker 算法的 *B* 空间中。

具有垃圾收集的几何形状的例子

附录 B 中介绍了用本章描述的符号习惯用法绘制几何形状（Shape）包的例子，演示了增量装入能力，并使用了垃圾收集的高级格式。垃圾收集技术不是基于引用计数，而是按照“标记和扫描”思想以及 Baker 算法完成的。

垃圾收集算法检查对象在现存对象 Shape 样本中的列表，寻找每个 rep 域，确定是否指向 Triangle 对象（应用 type 成员函数并检查它的返回值与 Triangle 样本的地址是否相等）。对每个这样找到的对象，设定“标记”位。扫描完成后，该算法按分配 Triangle 固定向量步骤进行，查找这些“标记”位未设定的对象。如果一个对象的“标记”位和“使用中”位都已复位，那么 Triangle 的析构函数就被调用清除此对象，并且它的“使用中”位复位。最后，该对象的“标记”位复位，以准备下一个垃圾收集周期。

这种算法首先循环处理所有 Triangle 对象，继之是所有 Line 对象，然后是所有 Circle 对象等等，直至将 ShapeRep 导出的所有类处理完毕为止，然后该序列再重新启动。在高层次上，该算法依次应用于每个应用域（如 Shape，Collection 及其他）。循环回收的调度是在实时环境中执行的，并需要调节每个应用。用 Baker 技术甚至可以做到增量垃圾回收。

我们考察一下附录 B 中该算法的实现细节，类结构我们在前面已描述过了，具有继承层次结构，如图 9.1 所示。所有细节的实现都在 ShapeRep 导出的类中，而所有用户可见的是 Shape 实例。

有些全局数据结构由静态成员函数 Shape::init 初始化。这个初始化是通过显式代码产生的，而不是通过缺省机制提供 C++环境，其目的是能够控制初始化的次序。Shape::init 首先初始化两种对象列表，一种是命名为 allShapes 的对象列表，跟踪所有 Shape 实例的建立；另一种是命名为

allShapeExamplars 的对象列表，它跟踪所有由 ShapeRep 导出类所建立的样本对象。

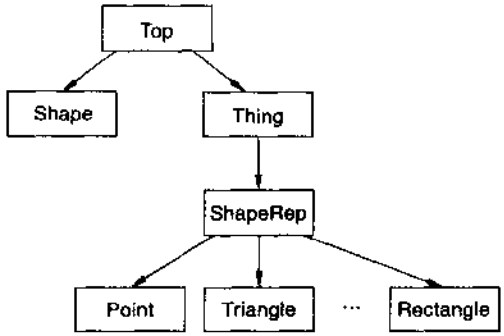


图 9.1 符号 Shape 类结构

Shape::init 的下一个引用由 ShapeRep 导出的每个类的 init 操作，依次构造它们各自的样本对象。每个样本记入通过 Shape::register 来创建 Shape 的日志，在 allShapeExamplars 列表中储存指向每个样本的指针。这就是为什么说 Shape 列表在样本初始化发生之前被初始化是如此重要的原因。这种初始化模式的缺点是，Shape 必须具有所有 ShapeRep 导出是存在的编译时的信息。

当初始化做完之后，用户会产生创建一个 Shape 的需求，使用自主类属构造函数习惯用法：

```
Shape object = (*shape) ->make(p1, p2, p3);
```

make 操作在 Shape 样本中被引用，产生与用户提供的参数相一致的对象。用户提供定义形状点的集合，并且如果指定一个形状（Shape）的点数没有二义性，则可以提供一个样本指针。例如，两个点可以定义一个矩形或者一条线段，所以第三个参数必须有所选择。3 个点所默认的是三角形（triangle），并导致以上的引用。

注意，上述 make 的引用能够用另一种方法来表达：

```
shape ->operator ->() ->make(p1, p2, p3)
```

即，Shape 样本产生它的内部表示指针，指向 ShapeRep 实例，作为对 make 应用的操作数。在 5.5 节的信封/信件习惯用法中，所有内存管理（包括 make 操作）均用信封来处理。这里，我们利用信件内的内存来管理。因为 make 是在信件类中的虚拟函数，所以它的新版本能够使用本章早先已描述过的技术进行增量装入。

这种引用调用了 ShapeRep::make，是一个重载函数。在 ShapeRep 中有两种 make 函数，第一种被称为对象期望类型的样本指针，除了坐标之外，还指定了对象表示的细节。第二种 make 函数仅用坐标作为参数，并合理地假定默认的也是所期望的形状是何种类型：它简单地调用具有合适参数的更通用的 make 函数。采用以上的引用，第二种 make 函数的引用是：

```
Shape ShapeRep::make(Coordinate pp1, Coordinate pp2,
        Coordinate pp3) {
    return make(pp1, pp2, pp3, triangle);
}
```

这依次引用：

```
Shape ShapeRep::make(Coordinate pp1, Coordinate pp2,
```

```
        Coordinate pp3, Thingp type) {
    return ((ShapeRep*)type) ->make(pp1, pp2, pp3);
}
```

依次产生具有 3 个指定顶点作为参数的 Triangle::make 引用：

```
Shape Triangle::make(Coordinate pp1, Coordinate pp2,
        Coordinate pp3) {
    Triangle *retval = new Triangle;
    retval ->p1 = pp1;
    retval ->p2 = pp2;
    retval ->p3 = pp3;
    retval ->exemplarPointer = this;
    return *retval;
}
```

首先，这个 make 函数使用 new 来创建新的 Triangle 对象，然后把合适的数据填入三角形顶点。对象的 exemplarPointer，即它的类型域，被填入指向 this，即指向三角形样本（this 在这个上下文中与全局三角形样本指针 triangle 具有相同的值）。最后，return 语句扫描新对象，返回到调用者。return 导致 Shape 构造函数 Shape(ShapeRep&)调用转换合适的返回类型的构造函数。

当 Triangle::make 调用 new 以获得新的 Triangle 对象时，Triangle 自己的 operator new 就被引用了：

```
void *Triangle::operator new(size_t nbytes) {
    if (poolInitialized - nbytes) {
        gcCommon(nbytes, poolInitialized, PoolSize, heap);
        poolInitialized = nbytes;
    }
    Triangle *tp = (Triangle*) heap;
    while (tp ->inUse) {
        tp = (Triangle*)(((char*)tp) + Round(nbytes));
    }
    tp ->gcmark = 0;
    tp ->inUse = 1;
    return (void*) tp;
}
```

Triangle::operator new 从 Triangle 池中返回指向有效的 Triangle 的对象大小的内存块指针。开始时，poolInitialized 的变量为零，它在编译时被初始化。与 nbytes 相比，它有 sizeof(Triangle)值，并可以表示其差别，从而引起调用 ShapeRep::gcCommon。这种子程序既可以对垃圾收集使用，也可在特定情况下对程序开始工作（启动）时，或者当类产生转换时作为内存池而被初始化使用。在其他情况中，gcCommon 的初始化阶段对所有池中的内存分配位给出初始值。再从 gcCommon 返回，new operator 通过在 Triangle 内存池（指向 Triangle::heap）中的块按步就班地寻找 inUse 位是否为复位的块，并返回找到的第一个块。如何处理空闲表用完的代码留给读者作为习题。

所有最新分配的 Shape 对象遵循相同的模式。从已存的或者从一个形状（Shape）转到另一个形状的指定的新 Shape 的初始化，引用合适的重载构造函数或者分配操作符：

```
Shape::Shape(Shape &x) {
    Thing tp = this;
```

```
    allShapes ->put(tp);
    rep = x.rep;
}

Shape& Shape::operator = (Shape &x) {
    rep = x.rep;
    return *this;
}
```

因此，多重 Shape 对象可以指向由 ShapeRep 导出的公共类对象。

下面的计数指针习惯用法，没有 Shape 对象从堆中分配，这恰好避免了释放无用对象的担心。当 Shape 对象离开了作用域时，任何 Shape 对象就被自动清除掉；当任何这个类的对象被清除时，则任何 Shape 的类成员随之也被清除掉。在程序终止时，全局作用域的 Shape 被清除掉。我们仍然会担心回收存储空间，在 ShapeRep 导出类的对象被清除之后怎么办，这里，垃圾收集器就出来完成这些工作，所以大可不必担心。

垃圾收集可以在任何时间引用。在附录 B 给出的例子中，在程序的几个点上是人工引用的。更透明的是当每次对新对象分配时，就引用垃圾收集，虽然这需要付出一定的代价。另一个策略是当 operator new 不能够找到任何空闲实体时，则仅回收未使用的内存。垃圾收集阶段可以把时间进行分割，以便能在实时环境中增量回收内存空间。假如垃圾收集遇到大的实时中断，则会产生问题。执行标记阶段本身以及对单个信件对象执行授权扫描阶段时，垃圾收集可用 Shape::gc 来调节：

```
void Shape::gc() {
    Listiter <Topp> shapeIter = *allShapes;
    for (Topp tp = 0; shapeIter.next(tp); ) {
        ((Shape*)tp)->rep->mark();
    }
    Listiter<Thingp>shapeExemplarIter = *allShapeExemplars;
    for (Thingp anExemplar = 0;
            shapeExemplarIter.next(anExemplar); ) {
        ShapeRep *thisExemplar = (ShapeRep*) anExemplar;
        thisExemplar ->gc(0);
    }
}
```

第一个循环通过每个现有的 Shape 对象设定它的“标记”位（用 mark 成员函数）。由 ShapeRep 导出的所有类对象，用已存的 Shape 来引用，并打上标记。第二个循环是对信件类的每个样本（对像每个由 ShapeRep 导出类这样存在的一个样本）来执行，并由它来处理算法的分阶段扫描。

因为每个 ShapeRep 导出类样本管理所有的由动态对象分配的它自己的内存池，所以每个这样的类很容易定位并扫描所有它的动态分配实例。在这里所介绍的 Shape 样本中，该池是一个内存邻接块；但它必须仅在此样本才能够跟踪它的所有实例。

例 9.12：遍及类型分阶段的扫描内存回收。

```
void
ShapeRep::gcCommon(size_t nbytes, const size_t poolInitialized,
        const int PoolSize, char_p &heap) {
    size_t s = nbytes?nbytes:poolInitialized;
    size_t Sizeof = Round(s);
```

```
    ShapeRep *tp = (ShapeRep *) heap;
    for (int i = 0; i < PoolSize; i ++)  {
        switch (nbytes)  {
        case 0:      // 正规的垃圾收集情况
            if (tp ->inUse)  {
                if (tp ->gcmark | | tp ->space != FromSpace)  {
                    // 不要将扫描做下去
                    tp ->space = ToSpace;
                } else if (tp != tp ->type())  {
                    // 对象需要回收
                    tp ->ShapeRep:: ~ShapeRep();
                    tp ->inUse = 0;
                    printf("ShapeRep::gcCommon");
                    printf("Reclaimed Triangle object % c\n",
                    'A' + (((char *)tp - (char *)heap)/Sizeof));
                }
            }
            break;
        default:     // 内存区的初始化
            tp ->inUse = 0;
            break;
        }
        tp ->gcmark = 0;
        tp = (ShapeRep* ) (Char_p(tp) + Sizeof);
    }
}
```

扫描工作是由静态的、在 ShapeRep 中由命名为 gcCommon 的公用子程序来完成的（见例 9.12）。该子程序扫描每个对象的字节数（nbytes），由字节数可知，在每个对象中最近的垃圾收集遍数（poolInitialized），以及在池（PoolSize）中的对象数。这些参数可以提供给在导出类中的 gc 子程序，并依次调用无参的 Shape。

作为以前的引证，ShapeRep::gcCommon 既执行了内存初始化函数，又执行了内存回收函数。垃圾收集逻辑是在循环内 case 语句的零分支。这种逻辑存放了已经打上标记的对象，或者是已经在每个 Baker 算法中的“到空间”处的对象。这个空间成员关系的检验在实现中并不严格，但是如果增量内存回收是作为早期引证的话，则是重要的。“从空间”中未标记的对象被回收：它们的“使用中”位为零，通过 operator new 返回到池中。

注意，我们已把对象的清除（析构函数引用）和内存回收（垃圾收集）捆绑在一起。对象清除是有关应用的语义动作，而垃圾收集回收的是特定的资源，这些操作均发生在令人感兴趣又令人忧虑的许多应用之中：即在内存之中。它可能分别产生两种操作，但给程序员的人工操作带来不确定性，对不再使用的对象的清除增添了更加沉重的负担，有了垃圾收集回收就好多了。例如，如果一个对象带有打开 UNIX 文件的文件描述字，那么从单回收该内存的资源来说，则仅仅是回收与该文件有关的资源。为了避免在用户层面上整理程序加工操作，我们把两个函数联合在一起。

这种方法假定用户能够提供无限周期的延迟终结。为了使终结发生，用户能够在对对象停止了最近的引用之后，立即引用垃圾收集器。

9.6 基本类型的封装

在大多数的符号环境中，每种数据类型的实例都是对象，与接近于机器的C++类型（int，char，long，short等等）相比，它比整个类对象更为基本。我们可以要求使用这些类型的对象，就好像它们具有符号类型的灵活性那样。例如，我们要求它们已被垃圾收集了。我们也能够要求像具体数据类型一样，即类具有所有“实”类型的灵活性，但却没有赋予符号类型的特征。

对于每个在符号环境中所使用的基本类型，我们能造就一个新类，以此为基础，建立类库，该类库与它们的基本类型家族相类似。虽然这种建立令人生厌，但这样的类型创建在概念上并不十分困难。其中一个例子是5.5节中描述过的Number类的收集。对Number代码的较少的改变是基于信封和信件的——主要地，从Top和Thing导出合适的类——完全的多态性，垃圾收集成员可以使用int，double，long，就像使用更方便的习惯用法一样。

在符号习惯用法中合并固有类型的主要困难是如何把它们放到继承层次结构的应有位置上。Number的例子企图把所有数值类型的功能归入某一类，但是以不区分double和float，或char和int，或unsigned int和其他之间的关系作为代价。String应当是符号类型（正如第3章假设的），或者应当在Smalltalk层次结构之后放在ArrayedCollection之下，SequenceableCollection之下，还是Collection之下？在第6章中我们曾强调过，这些决策应当由应用分析和应用领域来驱动。

9.7 在符号习惯用法下的多重方法

让我们来考虑一下，如果在两个数字对象上操作，尤其是当操作数是不同类型时的情形。处理这种问题的传统方式是要求编译类型系统充分地了解并安排类型转换，但这种方法在动态运行时键入的情况下就会发生故障。所以出现的第一种解法是求助于符号程序设计。假定对程序员显示的每个数都作为Number类的实例，具有不可见的信件类，诸如Complex和BigInteger。Number对象在运行时改变类型的特征，所以编译程序不可能有充分的类型信息产生类型转换代码。这种转换和有效的添加操作的选择必须在运行时执行。

我们如何选择一种算法使之能够增加给定的数字对呢？一种方式是对第一个数给出工作的任意性；另一种方式是选择operator+，这是基于第一个操作的类型；如果使用虚拟函数，究竟会发生什么情况呢。如果所有的类型信息添加到每个类型上，则加重了每个类型的负担，而且数字类型也变得缺少自主、自含和独立性了。

从概念上看，我们选择一种基于两个操作数类型的算法。C++虚拟函数不支持，但有些符号语言支持。使用多重操作数的类型去选择成员函数或操作符，在运行时被称为面向对象语言嵌入到Lisp中去的多重方法。

在C++中模拟多重方法就要求习惯用法基于如第4章中所描述的对象类型。通常不鼓励类型域；虚拟函数是导出类实现变体之间选择的最佳机制。下面我们讨论虚拟函数为什么不是一个合适的解。

在5.6节我们看到，在LPF类上定义一个操作符，一个低通滤波器（LPF）：

```
Value *
LPF::operator() (Value *f) {
    switch (f ->type()) {
```

```
        case T_Data:
            myType = T_Data;
            cachedInput = f;
            return evaluate();
        case T_LPF:
            if (f ->f1() > f1()) return this;
            else return f;
        case T_HPF:
            if (f ->f1() > f1()) return new Notch(f1(), f ->f1());
            else return new BPF(f ->f1(), f1());
        case T_BPF:
            if (((Filter* )f) ->f2() < f1()) return f;
            else return new BPF(f ->f1(), f1());
        case T_Notch:
            cachedInput = f;
            return this;
        }
    }
```

需要注意的是，对这种情况的分析可以变换到虚拟函数定义的聚集之中，也就会产生公共的虚拟函数，以扩散到大量的滤波类型，并且其代码无论如何也不能简化。

Filter::operator()(Value*)的语义是返回一个值呢，还是返回几个不同滤波种类之一呢，这要依赖于上下文来确定。上下文包括引用哪一个成员函数的滤波器的类型以及作为参数传递的对象类型。即存在输入滤波器，也存在输出滤波器以及要同时确定使用算法的类型，于是就产生了是什么样类型的结果。在 5.6 节中所介绍的例子，逻辑打包的工作是通过 Filter 导出类来分发的。以上 LPF 类的代码的这种工作分配给输出滤波器。如果采用虚拟函数方法，而且 Filter::operator()(Value *input)调用输入滤波器的成员函数来完成此项工作，那么智能处理该操作是在输入滤波器中完成。

这两个方法中没有一个方法能够全部满足要求。我们要求作为不单独属于一个类或另外一个类的混合操作来处理这些需求。这使人想起在 3.3 节描述过的使用全局重载的 friend 函数：

旧的：

```
class Complex:
public:
    Complex operator + (Imaginary):
    operator double();
    ...
    ...
    };
```

新的：

```
Complex operator + (Complex c,
    Imaginary i)  {...}
class Complex {
friend Complex operator + (
    Complex,Imaginary);
public;
    operator double();
    ...
};
```

这里，操作符在编译时从说明自变量类型中挑选；没有在运行时根据实际自变量类型这样直接方式来选择函数。这就是为什么我们要用多重方法的缘故。

让我们首先看看 Number 类这样一种简单的例子。为了实现 operator+多重方法，我们利用添加的可交换性，把专用情况减少一半数目。对类型域而言，我们使用虚拟函数 Top::type。虽然它是虚拟函数，但它作为公共的、枚举类的类型域而出现。

我们开始把新的成员函数添加到类成员和它的每个导出类中：isA(const Top *const)。当在类 *A*

的对象上引用时，这个函数返回为真，如果 B IS-A *A* 是真的话，则类 *A* 所带有的自变量指向类 *B* 的对象，这样就确定了两个操作数中哪一个更通用，并且因此获得该操作的控制。

我们也添加单一的 promote 成员函数，以替代早先样本的转换操作符。promote 成员函数把 Number 类型的自变量转换到为引用 promote 的对象类型中。给定的那个 promote 调用执行所有转换，它不必在每个信封类的特征中重载 operator*。在传递参数类型上不一定要用开关语句，因为每个信封的 operator+可以假设总能接受它的类的自己类型的参数：

```
class Number {
public:
    ...
    virtual Number *type() const;
    virtual int isA(const Number *const) const = 0;
    Virtual Number promote(const Number&) const = 0;
    Virtual Number operator + (const Number&) const = 0;
};

class ComplexX:public Number {
public:
    ...
    int isA(const Number *const n) const {
        return n.type() == complexExemplar;
    }
    Number promote(const Number& n) const {
        // 总是返回 Complex
        if (n.type() == imaginaryExemplar) {
            return Number(0,n.magnitude());
        } else if (n.type() == integerExemplar())  {
            return Number(n.magnitude(), 0);
        } else ...
        ...
        }
    }
    // 这个操作符一定只处理 Complex
        Number operator + (const Number&) const;
};
class Imaginary:public Complex  {
public:
    ...
    int isA(const Number *const n) const {
        return n.type() == imaginaryExemplar | |
            Complex::isA(n);
    }
    // 不升级：对 Imaginary 不需要升级
    // 这个操作符一定只处理 Imaginary
    Number operator + (const Number&) const;
};
```

我们用添加全局重载 operator+的方法以提供多重方法，使编译程序适用于任意两个 Number 的定义：

```
Number& operator + (const Number &n1; const Number &n2);
    throw(NumberTypeError)
{
    if (n1.isA(n2))  {
        Number temporary = n2.promote(n1);
        return n2 + temporary;
    } else if  (n2.isA(n1))  {
        Number temporary = n1.promote(n2);
        return n1 + temporary;
    } else  {
        throw NumberTypeError();
    }
}
```

注意，这样做提供了在 5.5 节中所提方法的一些优点。成员函数更具内聚力：对于给定类型定位的所有升级，以及所有双目算术操作符均在同类型的操作数上工作（无须重载操作符）。在类成员函数定义中不再需要 switch 语句。

这里我们大致阐述了这种方法，同时还介绍相关的符号语言。在一些 Smalltalk 实现中，如果两个操作数是相同类型的话，则特殊的数字类可直接处理算术操作。如果类型不一致，则对 Number 消息的处理必须转换和升级，并最终发送正确的子类方法。如果我们遵从 Smalltalk 模式，则全局重载就变成

```
Operator + (const Number&, const Number&)
```

尤其是

```
Number::operator + (const Number&)
```

有些 Lisp 环境实现多重方法是作为重叠虚拟函数来调用的，动态地寻找方法的路径，用多重自变量的类型来确定。

练　习

9.1 改变 Top::update 和其他代码，其目的是使新的虚拟函数能够添加到信件类中。注意，类的原始的 vtbl 在内存中是静态分配的。假定编译环境能够把在 vtbl 底部的新函数安排成实体种类。从信件类中继承的隐含的限制是什么？如何把虚拟函数添加到信件类中？

9.2 添加成员函数 Thing::backout(Thing*)，导致类的数据布局复原，并使虚拟成员函数复原到最近的更新之前。

9.3 修改附录 C 的程序，使垃圾收集执行每个新对象分配。

9.4 为了扩大垃圾收集，需要改变附录 C 的垃圾收集算法，使任何 Shape::gc 的单一引用仅从 ShapeRep 的一个导出类中产生垃圾收集。

9.5 编写一个内存分配系统，其中所有的类有相同的尺寸，并共享相同的内存池。同时，在相同的垃圾收集周期下进行管理。在池中类的映像应该在运行时执行。

9.6 重写 5.5 节中的 Number 类，并遵循符号习惯用法的构造。

9.7 编写基于 3.5 节中 String 的符号 String 类。

第10章

动态多重继承

在C++中，继承的主要目的是获取两种相似的应用抽象之间的联系，其中，一种应用是另一种应用的特定情形。继承结构可以看做是分类树，因为这些分类树通常有严格的层次结构，所以我们很自然地把层次组织结构施加到复杂系统上。

在实际活动中，要求一个类从多个父类继承一些特性，或者一个对象在同一时刻扮演多种角色，这是常见的事。例如，我们可以设计一种组合窗口的文本编辑（EditWindow），它具有专用的窗口技术的行为（XWin或MSWindow），它所产生的新类就同时具有两个父类的行为。

C++支持静态多重继承，即一个类可以具有从多重基类中继承行为的能力。但是，继承结构是在编译时确定的，并且许多真实世界的例子告诉我们，在运行时必须确定上下文关系：多重类如何组合创建一个为应用服务的新类。例如，如果一个系统包含不同的Window类的“套件”，我们就可能要求动态地创建合适的Window导出类，然后再将其实例化。在寻找环境中的命令行选项和其他可用数据之后，有一种程序可能会要求把EditWindow和XWin合并成一种情况，并且把EditWindow和MSWindow合并成另一种情况。两者取一就必须在编译时预先说明所有可能类的组合，与此同时，程序就从在运行时建立的窗口的可用类中进行挑选。类的组合爆炸导致了所有可能窗口类型的组合，因而引起了不必要的复杂性和令人乏味的软件结构。

多重继承已经给出一种在对象生命周期中扮演不同角色的对象机制。考虑一个SeaPlane类，它是从Plane和Boat派生出来的。这样的例子经常作为多重继承的经典而被引证。但是，SeaPlane的结果抽象不会在任何给定时间内显示其父类行为的组合：当在空中时，扮演飞机的角色。当在水中时，就扮演轮船的角色。静态多重继承不能获取这种联系的特性。例如，考虑Plane类和Boat类有一个成员函数，叫做mpq，它返回每加仑行驶多少哩（或假定这些都依次由Vehicle导出，而它本身有一个纯虚拟mpq函数）。这个SeaPlane的mpq成员函数必须查明对象在调用时扮演什么样的角色，并且在Boat部分和Plane部分的mpq成员函数之间进行选择，计算它自己的结果。管理继承的动态性是一种更加直观的满意的解。

当使用这种习惯用法时：如果要在运行时决定有关对象类型特征，或者需要引用多个父类时，就应使用动态多重继承习惯用法。动态多重继承是在Lisp中嵌入面向对象程序设计语言的弱模拟的一种参与概念。

综上所述，多重继承层次结构是我们所需要的，并且需要在运行时给定建立和修改这样的层次结构。本章通过一个例子来介绍一种模拟动态多重继承的方法，它是一种在层次结构中使用隔离类的内部的技巧。原始C++类的派生，虚拟函数和信封/信件习惯用法是使动态多重继承可能受限的组合技巧。

作为一个历史背景要指出的是，在这种多重继承技术成为语言的一部分之前，通常多应用 C++早期版本的多重继承。

10.1 多重技术窗口系统的例子

例 10.1 采用的是第 3 章中的多重继承的例子。在原来的例子中，其核心思想是 CursesWindow 和 XWindow 类的对象可以被实例化，然后整个地编辑就好像是 Window 类一样。其他的 C++程序使用可移植的 Window 接口编写。

例 10.1：对窗口技术使用继承性。

```
class window {
public:
    virtual void addch(char);
    virtual void addstr(string);
    virtual void clear();
    ...
private:
    Color curColor;
};
class Curseswindow:public window  {
public:
    void addch(char);
    void addstr(String);
    void clear();
    ...
prirate:
    WINDOW *curseswindow;
};
```

虽然有一种目的是为许多程序提供通用的 Window 机制，但由于编辑的需要，人们也希望能够剪裁 Window 类。EditWindow 是一种具有编辑特有属性的 Window：在屏幕的第一行和最后一行，源代码具有处理行回绕的能力等。EditWindow 应该是 Window 的导出类，但也应当具有像 CursesWindow 和 XWindow 一样的行为。那么我们如何构造呢？

对此问题我们考虑有以下几种方法来解决。一种是产生整个类的组合爆炸，导致以上讨论的类的组合为：

```
EditCursesWindow    EditXWindow
EditWindow          CursesWindow
XWindow             Window
```

这些类既可以由静态多重继承生成，也可以由一个类嵌入到另一个类中执行以类为名义的服务来生成。这似乎对添加新的窗口技术和增加其他新窗口功能性类是不太合适的，不具柔性的。另一种思想是避免全部派生，但有 Window，CursesWindow，XWindow 和 EditWindow 各自类（或对象）的类型。对象能够（如有必要）通过每个 friend 关系访问。但是，对 friend 的考虑具有一定的危险性，而且这种方法对封装的改变极为不利。

还有一种思想是从 Window 导出 EditWindow，并且从 Window 同时也导出 CursesWindow 和

XWindow。这就加重了带有 EditWindow 类的多余东西的 Window 用户的负担，并且势必造成不只是编辑方面的 Window 使用的退化。

替代的办法可以使用信封和信件类的变种。但要保证是从 Window 的 CursesWindow 和 XWindow 的原始状态导出，以使 Window 能作为类属窗口的接口而被开发。EditWindow 类包含了一个指向 Window 的指针，其中在运行时确实有既指向 XWindow，又指向 CursesWindow 的指针（见例 10.2）。所有 Window 在 EditWindow 上的操作都引用 Window 的（虚拟）成员函数，使用存储内部指针作为传递 this 的变量。

例 10.2：多态性另外一层的间接层。

```
class EditWindow {
public:
   void addch(char x)       {window ->addch(x);}
   void addstr(string x)    {window ->addstr(x);}
   void clear()             {window ->clear();}
   ...
   EditWindow()             {if (this_is_an_X_environment) {
                                 window = new XWindow;
                                ...
                             } else {
                                window = new CursesWindow;
                                ...
                            }}
private:
   Window *window;
   short topLine, bottomLine;
};
```

这种技巧就是大家所熟知的动态多重继承问题。但是对于这种方法，我们增加了另一个新的观点：EditWindow 仍然从 Window 导出，所以在编辑中的例行程序正好知道 Window 可以在 EditWindow 上操作：

```
class EditWindow:public Window {
   ...
};
...
Window *screen = new EditWindow;
screen ->addch('A');
```

这就是信封/信件习惯用法的变种。

当然，这里，EditWindow 类决定了层次结构的状态。对 EditWindow 构造函数添加自变量就允许建立的 EditWindow 更详细地指定导出的层次结构：

```
enum wType { Apollo, Curses };
EditWindow::EditWindow(wType t) {
   switch (t) {
      case Apollo:
         window = new XWindow;...
         break;
      case Curses:
```

```
            window = new CursesWindow;...
            break;
        }
    }
    ...
    Window*window = new EditWindow(Curses);
```

最后的调整是必须的。按照这样的方式，派生必须执行，即 EditWindow 的构造函数实际上给基类（Window）构造函数带来了空操作。如果基类（Window）构造函数被执行了，则可在屏幕上建立了一个窗口。窗口应该建立在 EditWindow 的左边。在某种意义上，调用了 CursesWindow 和 XWindow 的构造函数，并且把返回值放到 Window *指针的私有数据中。该构造函数的技巧是使用缺省的自变量来运作的：

```
Window::Window(bool doConstructor = true) {
        if (doConstructor) {
            ...
        }
    }
    ...
    EditWindow::EditWindow() : Window(false);
...
Window     *ordinaryWindow = new Window;
           // 与"new Window(true)"相同
Window     *editorWindow = new EditWindow;
    ...
```

如果存在以上 Window 类的基类，则可能需要对它们的构造函数增加一些特定情况的工作。例如，当使用样本习惯用法（见第 8 章）要求特定的手段时，那就应从 Class 导出所有的类。Class 构造函数仍被调用而无论"if"语句是否有，当然，在执行窗口初始化软件中什么也未做。通常，这种基类的初始化是未受到损害，但应当仔细研究以避免意外事件发生。

10.2 防止误解的说明

这种方法不会像在语言中直接支持动态多重继承那样通用。它所提供的多重继承模拟能力，在许多实践和经验中均被证明是有用的。

作为该方法的局限性的一个例子，让我们再来考虑一下前面曾经讨论过的 Window 的例子。我们所展示的对象有效的导出链能够在构造时间上指定：

```
Window *window = new EditWindow (Curses);
```

这种设计没有把下面的合法的性质归纳进去，除非提前采用特定的方法：

```
Window *window = new Curses (EditWindowType);
```

这种导出的次序最好遍及整个实现名字隐藏的不同的层次结构中（例如，在 Curses 中有一个函数隐藏在 EditWindow 中，以替代其他方式）。以上的设计或许还可以用添加代码的方法来实现，但恰恰成为难以处理的一个例子：

```
Window *window = new  Window (EditWindowType);
```

若这个要达到有关 EditWindow 的信息只对 Window 类的人使用，则是很困难的。

此段代码若能够使引用多重导出类的主人获得一点点技巧就是值得的。值得欣慰的是，大部分技巧对这些类的用户是透明的。

练　习

10.1　C++多重继承有哪两种范畴？在应用中，这两种范畴有什么实际意义？

10.2　动态多重继承与静态多重继承之间有何区别？试用具体应用加以阐述。

10.3　动态多重继承习惯用法在什么情况下使用？

第11章 系统性问题

本书前面的章节用对象和类作为基本系统构造块，研究了工具、约定和习惯用法。面向对象程序设计在第5章和第7章展示了有效的重用和抽象工具；第6章讨论的结果搞清楚了设计哲理规则的自然表达式。面向对象程序设计与其他技术的融合可以促使实现上的可管理性、可扩充性和可维护性；附录A讨论了在实现层上混合范型的问题。

在本章，我们首先回顾一下，以拓宽视野。对象范型能够很好地处理大型系统结构方面的问题，但数不清的其他设计上的考虑，在系统结构背后也应值得注意。对象家族的例外处理、调度、封装和管理都在这些问题当中。本章研究这些考虑的子集。虽然本章与第6章可以融合在一起，但并不能包括设计考虑的整个领域，它只覆盖了程序设计语言所能提供的一些指南，其中C++有特定的机会和特定的需求以处理系统层问题。本章不包括很深层次的素材，并且所谈的观点也不是一成不变的，但可以作为项目的特定构造以及实践的很好的起跳点。

这些系统性构造可划分为两种范畴：与静态系统结构有关和与动态结构有关联的。静态结构不仅包括由第6章的类图所获取的联系，而且也包括了高层的抽象以及不同的范围。不同的静态结构反映了我们对设计、系统传递和封装方法的认识。

动态结构的观点不仅包括在对象图中获得了什么，而且也必须包括如何动态地获取结构的改变，就像一些传统表示那样，比如处理调度、并行操作和例外处理的情况。这样的考虑也反映了我们对应用中系统性能的鲁棒性的认识。本章依次从这两种观点来研究系统性问题。

对熟悉操作系统实现的读者来说，也许认为本章为了嵌入系统设计就把“对象范型看做是与操作系统相一致的”。实际上，今天许多共同使用的设计技术其关键都是用抽象单位（unit）来处理的：对象是面向对象设计的配对物。这些处理程序是调度的单位：在对象范型下，调度单位是什么？操作系统提供了诸如输入/输出、资源分配和容错的公共服务：在面向对象的程序中，提供这样的服务应该是什么样的框架？在这里所描述的一些方法能够用于或替代传统的操作系统在应用中所具有的专门的资源分配、同步技术以及服务请求吗？

11.1 静态系统设计

的确存在着比对象作用域还要大的东西。从抽象意义上来讲，合理地使用对象可以反映系统中的高层概念实体：一种能够在大型对象内部用封装相关对象聚集的方法构造任意深度的对象的层次结构，然后包装此对象，并把它们合并在一起，如此等等。

但是，这种描述存在一些问题。让我们来考虑自动驾驶软件的设计：我们需要像副翼、舵、引

擎以及在它们上面进行操作的抽象。我们可以把这些作为飞机对象的聚合吗？根据我们设计的经验，试问我们能指定好的、干净利落的直观操作，而这些操作又是什么呢？是起飞、巡航、倾斜飞行？还是翻转？从飞行控制系统观点出发，这些都不是基本的操作。因为这些操作中的每一项操作都是在系统中与其他对象之间进行成千上万次调节的结果，所以是不太容易作为设计算法或作为成员函数那种实现的东西的。这说明，高层次的抽象是必须的。

向系统添加单一功能至少要严格地映射并添加到相应的软件上。例如，对自动驾驶仪添加“圆筒棍”，就要求充分地把这种改变映像到已存的系统功能的实现代码上。功能上的改变必将遍及到设计和实现范型上。如果进程是组织的单位，那么大部分终端用户就要求跨越所有的进程。对象试图封装这种变化，因为它们反映了领域专家要求保持的不言而喻的抽象，但这要归就于此问题本身。应用结构和解结构之间的同构可以减少整个改变的影响，但这不是最好的办法，因为某些改变将不可避免地超越设计领域中一些主要的假定，并且只能够通过类的重构来调整。如果存在着什么保证，那么大的改变就会有大的影响，小的改变就会有小的影响：这是最少意外的原则。

这里，我们准备探讨了对象层上的几种抽象机制，其中的两个（事务图和框架）是开发过程中前端的静态设计工具。而模块、子系统和库是组织和编写软件设计的基本要素。当然，也可以作为向下设计的输入。

事务图

虽然面向对象设计集中在系统的自然构造块上，但要求我们能够根据运作的功能直观地认识一些系统，即使这些功能不是作为基本操作来实现的。考虑一种图形设计语言的编辑程序，它用方框、圆和其他形状类型作为类，并用线来连接这些形状。我们要求编辑程序支持 moveline 命令，这样，线的重定位就会把这条线连接到形状的终点，而其他的线则根据形状的轨迹连接到相应的位置。

moveline 命令由对象合适的数据结构和成员函数代码组成。这种成员函数代码在什么地方编写呢？移动一条线就产生了在 Mouse 对象、Screen 对象、单个形状对象和 Line 对象之间的相应的动作，所以，moveline 为公共目的一起运作的对象的调整要比单一对象上意义明确的操作少。这种复杂的交互就称为“商业事务”意义上的事务。对象的聚集提供了事务，对象聚集的联系称为**机制**。事务并不是任何对象的必然的职责，而是作为对象与运作系统功能之间交互的结果。

许多大型的活动都归入事务这一范畴，这比作为成员函数的处理更好。在电信技术学中所描述的如呼叫等待、呼叫转移以及账单的特性，不应作为对象和对象上的成员函数来看待，但却可以将其作为事务。下面是航空事务的一个例子。

当分析高层事务时，例如，电话转发或飞机着陆，面向对象技术应该放在一边吗？如果每个事务当作过程来处理，那么又如何逐步进行细化（功能分解）呢？刻画系统资源的对象仍试图在长期运行中最好地封装对象的变化。如果放弃面向对象技术，并允许事务驱动系统结构，就会导致整个系统结构的蜕化。采用过程方法就会在错误的地方转发消息；大多数过程都依赖数据结构的内部细节，而这种数据结构又与外部有关联。这就使得对代码的理解和进化都很困难。驱动对象方法分发了单个事务的消息——但如果设计者了解到事务需要做什么的话，就会很清楚需要引用什么样的对象，并且变化的轨迹也容易识别。

事务图就是这样一种设计工具，它能够帮助我们标识合适的对象以及在应用中标识它们之间的关系。通常，用户要与设计者在商业事务层面上交流，并对成员函数进行抽象，对用户来说也不要

太过于抽象，从而使其在理解上相对容易些。但是，当对象通过具体的成员函数进行交互时，这些商业事务就知道系统该做些什么了，并且获得适应用户的系统性功能观点的映像，这一点是很重要的。

事务图就描述了这样一个完整的功能性的机制。这种机制可以是整个系统，也可以是子系统。子系统的描述如下。

在事务图中每列都是一个事务或者一个机制。每行对应于一个对象。行和列的次序是很重要的：逻辑相关的功能和抽象应当在表中紧紧相邻。在逻辑上应当认为左边与右边相邻接，而顶边与底部相邻接。一种贯穿事务图从左到右的移动，在系统周期内就会穿过连接的事务处理。图 11.1 就是这种真实的例子。

商业事务

对象	起飞	爬升	巡航	右转	左转	下降	着陆
襟翼（着陆阻力板）	升起						降下
左副翼				降下	升起		
右副翼				升起	降下		
左升降机		升起		升起	升起	降下	
右升降机		升起		升起	升起	降下	
舵	右			右	左		
油门	增加	增加		增加	增加	减少	减少

图 11.1 飞机系统的事务图

表的元素是一种事务执行给定对象的成员函数。我们要查找事务图中的模式：类似的功能的聚集应用于相关事务的有关对象上。这些相似性指向了对象结构中的相应的地方，并能够将其分解到公共基类中。例如，我们知道，当飞机爬升时，两个升降机同时提升。这就暗示出两个升降机有某些共同之处。Elevator 类可以作为两个升降机种类的基类来使用，并且含有 climb 成员函数。当飞机爬升时，知道升降机需要执行什么动作。

我们也看到，在飞机左右翻转时，需要调节油门。这就提醒我们，某些全局函数应当调用左右转向舵，包括不属于单一对象的逻辑上公共的两个事务。

若能采用更全局的观点，我们就能够运作得更好。因为左右升降机的行为是同一的，所以我们无须在设计层上处理它们的差异。每个升降机是公共类的单个对象，并且我们上面标识的 Elevator 类可作为某些公共的提升操作，所以 Elevator 类能够处理全部的操作。两个升降机各自的行可以合并成一行。

有些成员函数是不作为聚集来考虑的，比如在校正螺旋桨旋冲的起飞期间，舵上的 right 操作就可以用直接的单一成员函数替代，把其他成员函数合并到事务中去。

事务图的侧重点应当是在事务的表示上而不是在形式上：它是一个可以帮助我们标识模式和集合的工具。这种方法很像在序贯硬件电路设计中（称为 Karnaugh map）所使用的形式工具。

模块

模块是一种结构配置的单位。C++只支持与 C 相同范围的模块，即通过代码的组织进入各自的

源文件中。若一个模块能够包含单个类，也能够包含好几个类，同时还能够输出其他类型和约束。

可将模块划分为两个部分：一部分是用户使用的符号输出；另一部分是私有实现。模块的输出部分在头文件中指定（调用“#include”文件），通常有.h 或.hpp 后缀。大多数私有部分是在代码的源文件中（带有.c 或.cpp 后缀）。某些私有实现也放在头文件中。

头文件应当由满足类型系统和表示的基本说明组成。在头文件中的典型的说明包括在模块内使用类的说明以及定义在模块内的并被用户访问的全局变量名的说明。其他与模块有关的类型定义和枚举也出现在头文件中。有些定义放在头文件中是有用的，尤其是符号约束定义，无论是全局的还是在类的作用域内的。

代码源文件包括了模块的类成员函数的定义以及全局函数和全局对象的定义。

代码源文件的内容应当对模块的用户不可见；头文件是模块用户获取访问模块接口的#include。这种建立可以使内联（inline）扩展函数陷入困境：它们最好分别在各自的代码源文件中加以维护，每个说明均使用 inline 关键字。如果函数是内联扩展，则该文件可以在模块头文件的末尾上呈现为#include，或者如果不希望内联扩展，则可以分别加以编译。例如，Stack 类，Stack.h 可以如下编译：

```
#ifndef  _STACK_H
#define  _STACK_H
class Stack {
public:
   int pop();
   ...
};
#ifndef inline
#include "StackInlines.c"
#endif
#endif
```

并且 StackInlines.c 可以如下编写：

```
#ifndef_STACKINLINES_C
#define_STACKINLINES_C
#include "Stack.h"
inline int Stack::pop() {
   ...
}
#endif
```

而 Stack.c 包含了没有内联函数的 Stack，可以这样编写：

```
#include "Stack.h"
#ifdef inline
#include "StackInlines.c"
#endif
Stack::Stack() {
   ...
```

现在，当编译此系统时，采用定义 inline 宏为空串的方法，由产生每个 Stack 成员函数的单一的紧耦合的拷贝来替代多重内联表达式：

```
cc  -Dinline = " " *.c
```

子系统

子系统是代表自主可交付的模块的聚合，在功能上是丰富和完整的产品。例如，一个组织或公司能够签订子系统的设计和交付的合同。

在早期系统设计中，某些类型的抽象的初始可以作为子系统实现的很好的候选者。优秀的设计者在给定的领域中应拓宽抽象的作用域，以适应大部分应用的需求；当拓宽发生时，一定要标识模块和模块的聚合，以便将来可能需要重用这些应用。这样的抽象对子系统的封装是有价值的，所以要尽力为现有项目撰写文档，组织寻找可重用的代码。

子系统经常实现这样一种机制：类的聚合提供超越成员函数的高层应用功能。运行时支撑环境、数据库管理和网络接口，这些都是子系统的例子。

一个类可以作为静态成员函数的功能聚合群体来使用，并具有像子系统一样的结果抽象行为。这些函数通常比原始（非静态）的成员函数彼此的相互捆绑更松散些，并且不需要多态性处理。子系统通常不对应于单一的资源，这种单一的资源是作为面向对象分析实体或类型来描绘的。子系统是单独的实体，仅仅是系统中的许多子系统中的一个。一个子系统通常是概念上的，它具有不同的和去耦合的内部状态，并且比对象有更少的语义亲合力。我们来看下面的一个操作系统接口的一例子：

```
class OperatingSystemInterface {
public:
    static int reboot();
    ...
    class Process {
      ...
    };
};
```

但是，在一个层次上的子系统可以考虑在另一个层次上的对象做什么，而且静态函数可以是在另一层次上的“实例”（非静态）成员函数。无论是抽象的是对象还是子系统都主要依赖于其内聚力以及与应用领域有关的直觉。如果从系统观点来加以描述的话，那么这样的资源应当是对象，以替代子系统，而且应当避免静态成员函数。

把静态成员函数封装在类中可能相当于 Ada 的包。另外还有静态函数，诸如一个类能够作为其他类、枚举和常数的输出。

框架

框架定义了可定制和可扩充的子系统或机制。子系统设计是由类的集合封装的，虽然实现可能仅仅是部分指定的。框架抽象基类舍去了应用的一些成员函数的定义。面向对象框架的例子是 XToolkit，MacApp 和 Smalltalk Model-View-Controller。

框架提供了与操作系统相关的公共服务程序。与大多数操作系统不同，框架提供了某些领域的特定的服务。框架超越了操作系统的单一服务，刻画了应用的主要抽象；框架的用户所提供的代码填入到抽象之中，使之构成了完整的系统。

C++模板是定制现有应用参数化软件的方式之一。框架试图进行扩充以获得更加通用的抽象。通用的框架能够作为抽象基类的聚合来构造，当给出用户定制特定的语义时，就限制了设计的“形状”。一种有用的框架习惯用法是基类抽象的公共成员函数，它舍去了私有实现的细节，虚拟函数由

用户导出类重载。框架有很少的基类，如果有，也都是公共成员函数（见例 11.1）。大多数公共操作在基类中采用通用方式来处理。基类成员函数依赖于子程序实现的序列，此子程序是作为导出类中的私有虚拟而被提供的。

框架能够作为分布式库的封装。

例 11.1：框架的构成。

```
class TaxFormFramework {
public:
   TaxFormFramework(int numberofLines) {
     ...
   }
   void draw() {
     makeField(loc1, taxpayer.name());
     makeField(loc2, taxpayer.ssn());
     ...
     addFieids();
   }
private:
   void makeField(Point, const char *const);
   // 导出类必须有它自己的 addFields 版本
   virtual void addFields () = 0;
   Point loc1, loc2, ...
   ...
};
class ScheduleB:public TaxFormFramework {
public:
   ScheduleB():TaxFormFramework(14) {}
private:
   Point locA, ...
   void addFields() {
      // 对 ScheduleB 添加一行
      makeField(locA, mortgage.income());
      makeField(locB, other.income());
      ...
   }
   ...
};
```

库

库是一种从应用中收集和挑选所需要的一片（段）软件的聚合的组织，它类似于子系统，但它是比软件组织单位或抽象单位更好的封装单位。

库通常用两部分来封装：一部分是对象的代码；另一部分是头文件的收集。封装是用库中的抽象部分的方式来构成的，而无须造成库中未使用那部分的开销。

每个库应该是围绕主题来建立的，就好像通用逻辑库中有许多书和其他历史种类的资料一样，并且农业库有在生活科学中如何种植农产品这样的参考资料。一个库就像一个可以分别购买到商品的交付子系统，但它更像一个书库。

每个库确实有一个规则集合，它必须统一地应用于所有的抽象中。内存分配技术的一致性、例

外处理策略以及类外延机制（使用模板或导出类）使库变得易于理解和应用。如果所有库都以这样的相关性和一致性环境呈现，那就太理想化了。另外，库在实践中的实施也是很困难的。简单接口类的聚合能够在已有库的类的基础上与局部接口标准相一致，或者投其所好。调整内存分配技术则是更困难的事，而且大多数独立于应用的库仅依赖于语言直接支持的分配和解除分配。用户希望使用更高级的技术，例如在第九章描述过的垃圾收集技术，就应当以这些库的类为基础来建立处理类，以管理它们的内存。值得庆幸的是，大多数这样的转换不需要对原始库进行编辑或者重新编译源文件。

当创建库与库之间的依赖性时，就必须进行特别的维护。处理相关库的类之间的转换应该使用约定和在第 3、4 章中介绍过的经验规则。注意，必须小心创建类的模板库，以避免轮流使用模板的实例，而造成冗余的拷贝。在第 7 章中我们讨论过一些方法，用来减少这种代码的冗余。

只要遵循这些指南，库就能够封装各个具体数据类型代码，也能够封装基类聚合或者整个框架。鉴于一致性缘故，库也可封装和分发子系统的代码。但是，通常情况下一个子系统可以把整体代码收集、合并到一个应用之中，所以选择装入库的封装的好处是很少对子系统库做训练；单一重定位文件也能很好地交付使用。

大多数库的设计是对软件重用原则的一种注释，并且在这里所有第 7 章介绍过的原则和指南都可使用。好的程序设计技术和软件组织一定可以简化库的组织和管理，这是软件设计技术的大方向。目前已有许多很好的例子。

11.2 动态系统设计

大多数设计技术都集中于静态设计。动态系统的特征要反映在一些设计上的考虑中，但系统动态的整个处理要求反馈到原型或早期的实现中。这些动态问题包括运行时各个实例之间协作的模式、系统实时约束的调度规律的能力以及分布式处理的作用等。这些都是所有传统操作系统所关注的问题，但需根据面向对象原则加以修正。

调度

调度是重要的设计问题，因为作为调度单位的进程支配着许多软件模块。调度接受非显式的 C++语言支持，这并不意味着调度问题是 C++以外的东西。C++语言的创建，它类似于 Simula 67 支持抽象事件模拟，而且 C++用任务库来支持事件的调度。在当前的 C++环境中，许多安装程序都连续地使用这种任务库的子孙后代。

为了更好地讨论调度问题，有几个术语需要解释一下，作为我们共同理解的基础。这就是本章所使用的术语——进程、多进程和并行进程。

进程——进程是一个调度单位，也可以是保护单位、地址空间或者名字空间的单位。被调度的结构单位通常是程序。

多进程——多进程是基于进程的作为调度单位的调度技术。多进程可以在单进程的基础上产生，对每个进程呈现出虚拟处理器的感觉。一个单处理器的多进程环境也称为计时-共享环境。在这样的环境中，进程之间不存在真正的异步，虽然“伪并行操作”模拟了异步的结果。

并行进程——并行或分布式进程引用多个处理器，或者更精确地说，引用的是多控制并发进程

（程序计数器）。大多数多进程规则均应用于并行操作，但是，补充的考虑必须给出真正的异步；它不在单处理器情况下发生。

软件应用覆盖了实时和调度的非常广阔的领域。不可能用一种调度方法就能应付所有的应用。语言对多进程的支持，也很难适合这么宽泛的应用领域，一种很狭隘的构想调度模块也很难适用特殊的应用。例如，某些应用可能需要多进程而不需要并行操作（例如，自主实体的模拟），而其他应用可能需要抢先调度技术。对分布式的支撑也必须是一个实例一个实例地进行设计。某些环境可能支持低层网络协议的并发，而其他环境却可能不全支持，并且仍可能通过共享内存来支持这种低层网络协议。这种不同的构造环境要语言提供支持是很困难的，既使惯用的构造能够支持单一的、可重用模块的并发。假定语言支持这种并发，但单个项目也需要很好地调节已有的模式（诸如 C++任务库）或者需要调整建立的全部新方法。

在第 6 章中我们注意到，对象范型下的重要的设计原则是力争在应用结构领域和解对象及类结构之间建立并行性。我们称这种特性为同构结构原则。在这两个世界中的“结构”是用相关行为的聚集来加以刻画的。那么调度的原则是模拟吗？

经验，再加上一点自我检测（监视），我们所能给出的回答是：也许吧。如果计算的结果模块是基本的，并且很清晰地反映了应用部分的特征，那么回答就是肯定的，因为它符合同构结构原则。比如，考察一下面向对象数据库的设计，其对象驻留在共享媒体中，并且它们可以被多个处理器访问。对象是同步的单位，并且为了保护数据的完整性调度机制需要作高层设计的考虑。这是直接与问题解有关的并行性的好的例子，这个例子出现在问题叙述的字典之中。

相反地，可能问题所处的地方不是固有的、并行的，而是把并行性建议作为解本身的一部分。这种情况的一个简单的例子是计算机所产生的动画图形渲染。另一个很好的例子是在电话网络控制系统中的电话呼叫抽象：当“呼叫”是单一概念实体时，其中一部分可能会驻留在一个紧挨着原始终端的电话控制处理器中，而其余部分可能驻留在与呼叫部分相关联的控制处理器中。这里，分布式不仅需要产生调度问题，而且也会改变结构。

在大多数应用中，是采用特殊的调度策略实现，还是采用单 CPU 或多 CPU 来实现并无多大区别。通常情况下，调度从属于实现工程，而不是应用本身的特性。某些应用并行性是固有的，而其他的应用必须遵循性能上的约束，强制在实现上采用分布式处理。与已有软件共存，或者使用特定应用的处理器（单处理器，图形加速器，算术处理器）也可能对解的具体的调度方法进行约束。许多在解领域的规格说明和应用领域约束之间的区别是人为的；许多应用领域的约束在问题结构中是隐含着的。

面向对象设计中的每个对象可以作为进程来处理吗？上下文的转换开销和大部分多处理器环境的分发消息的花费对处理调度单位的所有对象来说太高了。其中的含义是执行操作上下文转换，或者每次向操作系统发送字符串消息，以便寻找到另一个字符（换言之，Smalltalk 发送的消息，与处理器调度的关系只是松散地捆绑，并且还隐含了同步）。此外，贫乏的语义却适合大多数应用。多处理器的存在对给定的多进程产生一种运行在自己处理器中的感觉；无须指出这是在串和它的用户之间的错觉，而只需要用连续的程序计数器执行就可以满足要求了。

从另一方面来说：如果我们需要对进程进行抽象，那么进程是对象吗？回答是这种抽象是否构成了设计上的干净利落，语义上清晰的成员函数。如果我们要用进程来处理这些对象的话，那么回答是肯定的。考虑到具有成千个处理器的高度分布式处理环境，如果我们频繁地要求处理器计算输

入和输出，那么我们理想的要求是把调度单位作为体系结构分量来管理。这些对象可以是从基类 Process 派生出来的；我们称这样的对象为 Actors（施动者），似乎独立地扮演并执行自己的角色，而每个对象都有它自己的程序计数器。

如上所述，C++语言本身不支持 Actors 和其他调度规定。但是，运行环境能够把 Actors 放在 C++应用程序支配之下，以支持这样的模型。Actors 模型也非常近似于许多通用分布式的 C++环境以及流行使用的 C++任务库。下面是有关这方面内容的一个简短的描述。

Actors 把结构单位和调度单位合成了单一抽象。如果是处理两个各自的、松散耦合抽象的实体，那么对设计的约束就较少。如果基于进程的体系结构不复存在，那么结构单位、地址空间和调度就会是紧耦合的。与此相反，我们更希望调度结构减少对象，并通过几个不同对象的成员函数运行在有限的字符组的计算之中。这些字符组被称为线程[1]（Thread）。

C++任务

通常采用 C++库支持任务（task）作为调度的单位，这种方式近似于施动者（Actors）方法的模型。任务对象可作为调度单位，也可作为结构单位。任务对象也是名字空间的单位——因为它们有自己的作用域。C++任务包初始设计为抽象事件模拟，但 C++任务包也可用于任何调度应用所需要的非抢先调度中。

库类提供了多任务支持：

任务（task）：类的对象从 task 中导出，表示激活系统资源。每种行为仿佛有它自己的程序计数器，就好像运行在它自己的处理器上。task 对象的构造函数担任“主”子程序的作用。像进程一样，一个 task 对象有它自己的私有数据和它自己所控制的路径。一个任务可能会暂时停止用 object 资源状态的块控制，或者会撤回 n 个时间分段信号处理器所引用的 delay(n)。

对象（object）：类 object 作为被动系统资源的基类。这些资源既可以是在进行之中的，也可以是准备好的，而 task 能够在一种和多种 object 实例状态上中断它自己的执行。如果多个实例正在等待，则任务迅速恢复其中的一个，并把这个状态改变为“准备好”。

队列（queue）：任务相互通信是由通用目的的队列运作。队列具有头（head）和尾（tail）的属性，分别用 qhead 类和 qtail 类来表示。qhead 和 qtail 都从 object 导出，所以一个 task 能够中断在 queue 的头等待所提供的数据。

注意，虽然 task 可以彼此直接引用成员函数，但这样的引用是完全同步的，而且不采纳 task 的语义。在 Actors 的范型中，对象之间的交互通过消息接口来完成，而消息的内容则受到运行时解释其语义的接受器的检查。一个任务的接口不直接用另一个任务访问，所以类的特性在基于任务的系统设计中不是用图表来表示的。实际上，虚拟函数对 task 对象是无法使用的：多态性可以获得（用户定义）运行时消息内容的解释。

task 库提供了低层的同步性。首先，因为所有任务均运行在单一线程控制的单处理器上，所以应该是同步的。第二，值得指出的是，用户层活动的同步是通过在 object 和 queue 对象的中断机制中实现的。实现的细节对用户是隐藏的。

优先级驱动调度方案可以添加到任务对象系统之中。例如，它可以要求基于优先级的调度，也就是说既可以要求优先级调度，又可以要求在队列中包含优先级调度。优先级大部分用于 I/O 较为

1 线程是一个逻辑概念，不是一个物理概念。

集中的系统，其中外部活动可以触发高优先权的活动，把正在进行的计算并行运行。

当使用这种习惯用法时：任务对象对抽象事件模拟和其他希望模拟并行的单处理机应用是有用的。当调度单位是由应用本身的适当的抽象组成时，使用这种习惯用法就可以替代人工解。

线程

线程是对象的一个普通调度机制，它提供了 Actors 模型的一种选择。Actors 合成调度单位和结构单位，线程允许两种抽象作不同的处理。有人把调度策略作为进程模型来处理，则线程将会在相同的意义中以异乎寻常的方式出现——抽象的类对某些不精通功能分解、不具备这方面经验的人来说是陌生的。

图 11.2 演示了线程调度的过程。线程的执行是由“请求”或“激励”引起的。假设激励标号 1 来自于外部产生的消息。所有激励和请求都由调度对象来发送，映像请求对象的类型接受并引用它们的成员函数。大部分请求将使这种信息嵌入到对象之中，由另一个原始请求的系统对象来提供：在这种意义上说，这些请求表现得更像消息，或者类似于在 task 库的队列中的输入。

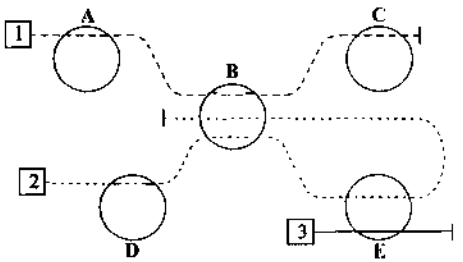


图 11.2 对象成员函数的线程调度

调度对象把激励消息 1 发送到对象 A 的成员函数，就开始执行。此成员函数调用对象 B 的成员函数，依次调用对象 C 的成员函数。由 C 的成员函数返回；最后控制立即返回到发送的控制点 A，这时该线程就完成了。线程从开始运行到结束不能中断：因为在资源块或收回处理器模块上没有代码，直至线程完成为止。出于策略上的考虑，系统的计时器用于确保给定的线程在适当的时间、区间内执行。调度启动的计时器在线程的开始处，并且当线程终止时，令计时器无效。如果计时器在线程执行期间就被触发了，则当前的线程可能认为毫无实际意义，并且可能会产生复位动作。

线程的执行可能对调度产生其他的激励或请求。当调度在线程终止处收回控制时，就会从前一个线程请求中产生新的线程，并启动此线程；如果没有这种新线程，则保持空闲状态，直到外部激励重新开始执行为止。线程 1 的执行可能已经创建了 2 的请求，并启动它自己的执行线程。

因为不存在显式中断，所以同步主要用线程来辅助。在线程之间，没有一个对象处于“部分计算”状态，因为它的成员函数之一因等待资源而被阻塞。

线程调度以正常的方式对分布式、多处理器的环境进行综合。任何单线程运行在一个处理器内；所有处理器之间的通信发生在从一种处理器运行时的支撑环境到另一个处理器环境的激励传送过程中。

线程能够用基于激励类型的优先权模式来驱动，其中调度可能基于激励的类型或任何其他合适的规则，而混合处理激励的顺序。

当使用这种习惯用法时：在实时应用中使用线程，对实时事件响应的时间是严格的。线程对分布式体系结构是有用的。这种体系结构需要在设计或部分实现的后期把对象捆绑在处理器上。

上下文

在计算机科学中，所有感兴趣的问题都归结为在名字空间中能够做什么样的简化。在面向对象设计中，大多数类和它的成员函数的名字是全局的，作为很多系统对象来处理。试图寻找对象、判

断改变的影响以及管理运行时的资源，在许多有顶层类的系统中，这项工作变得很困难。

如果我们检查一下静态系统结构的抽象，如类、模块、子系统、框架和库（以及稍微次要的事务），我们就会发现每个抽象对源代码的一些聚集都有直觉的联系。上下文描述了现存的、运行对象之间的联系，其含义就是静态抽象的联系，因此，对源结构来说是很松散的。重要的是承认这样的抽象是运行时管理的单位。同样的道理，进程是当今设计技术中流行使用的重要抽象。

大多数应用的实体都聚集到共用体中。即大多数对象只与少数几个对象相互作用。这些共用体经常作为局部聚集来处理，能够独立地初始化、管理和发送。一个对象的层次结构（在一个对象内封装另一个对象的实例）是这样聚合的一个例子。在层次结构之外的其他对象的聚合不能反映抽象的集合，也不能获取这些对象之间的联系。例如，我们很清楚 Mouse，Menu，Keyboard，Screen 和 Window 类之间是相关的，但它们并不是个个都属于单一类的抽象（虽然 Screen，Keyboard 和 Mouse 可以作为 Terminal 的聚合）。在某种意义上，这些共用体恰巧是子系统。但一个对象可能属于多个共用体：一个 Window 可以属于图形编辑所实现的对象共用体，也可以隶属于窗口管理的共用体集合。Telephone 和 Call 对象在电话呼叫中表示一种局部的引用。但是，任何给定的电话可以同时驻留在两个这样的共用体中，如果一方面使用了共线电话呼叫等待，另一方面又正在与另一个电话交谈。要以令人满意的方式表达这些联系，既不是“大型对象”也不是子系统。为此，我们为共用体杜撰了术语**上下文**。

每个上下文有一个服务器对象，其动作就像它自己的微型操作系统。对象仅通过服务器进行相互之间的访问，而服务器仅从上下管理中进行访问。该服务器函数是基本的，支持名字服务以及一些基本内存管理。名字服务函数映像对象的符号名，这些“符号”名可以是串，也可以是枚举的常数。它们使对象在上下文中能够相互访问，无须搅乱整个程序的名字空间。当一个对象生成时，它本身就可以存入到一个或多个服务器之中。

上下文在定义名字空间中使人想起传统的进程。其他传统的进程也能够合并到上下文中，诸如内存管理和错误处理（调度是比上下文更高层次的处理，在抽象层上更接近于处理器的映像）。所有对象为上下文而产生，或者从运行上下文内的代码中创建，可以用这个上下文的服务器来分配存储区。

贯穿名字空间的相互作用

分布式环境应当在不同的进程和不同的处理器中提供对象之间通信的公共支持。如果分布式通信集成在进程内的对象通信机制中，那么该环境可以支持分布式的透明性。对象上下文转换的开销也应当是很小的，用定制远程交互作为需求，使用单一的、通用的统一机制来替代，则开销是很高的。本节研究的 C++习惯用法提供了分布式的透明性以及通用性和有效性的平衡。

进程与对象再访问比较

无论体系结构是否使用进程，还是对象作为体系结构的主要抽象，该抽象必须相互通信。当一种抽象引起了对另一种抽象的控制（自动地或其他方式），我们就调用**上下文转换**。

在进程体系结构中，上下文经常转换引用消息。如果传送的控制流不如传送的数据那么重要，那么共享内存可以用消息来替代，无须花费消息的格式化开销。在这两种情况中，当从一种进程控制传递到另一种进程时，操作系统必须存入进程状态。这包含当从另一个函数调用时用一个函数存入正常的信息：使用该程序的计数器和各种各样的寄存器。操作系统也必须调度另一个进程：重新

存储它的状态，并且当分配到的那个进程被抢先（预占）时，就不再使用，或者恢复之处作为状态的改变或激励的结果（例如，信号的接收）。所有内部进程的通信产生了这些上下文转换的最小的开销，并且潜在地创建了消息。

如果对象代替了进程成为体系结构的单位，那么就必须有一个上下文转换机制。普通的 C++对象之间的上下文转换机制就是普通的函数调用。该调度是直接的，可以替代消息映像表。该函数调用栈协议，存入引用的上下文，返回恢复。将自变量压入栈中，替代消息中的传送。所有内部对象的通信都产生了单一函数调用开销。

每种方法都有它的优点和缺点。进程可以以任意次序调度，使之易于实现基于消息到达的次序用优先权来调度的模式。进程比对象更容易分布式地遍及处理器；函数调用机制用于对象之间的通信，不能扩充到多处理器配置的情况。通用性所付出的代价以及上下文转换和消息格式化的开销都能够被具体化。

在面向对象的世界中，我们希望尽可能地从直接函数的调用中获得好处，并为我们必须使用消息（分布式进程）付出代价。我们更希望尽可能地具有高度的分布式透明性。下一节我们将研讨这种折衷方案。

分布式透明性

系统的演变使处理器的数目可以增长，而对象也可以从一个处理器移动到另一个处理器。技术和需求的改变会导致强制改变对象的分割，使之可以跨越进程和处理器。这样的改变不仅出现在长期维护的整个航向上，而且要考虑在初始实现期间进行权衡。使用线程和特殊的处理以及主体类，这些对象能够以最小的努力使之能够远程跨越许多处理器。

每对对象之间的相互作用可以分为两个部分：请求部分和调度部分。每个对象也有两个部分：处理部分和主体部分，参见 3.5 节中的习惯用法。为了提供分布式透明性，处理的类对象能够激发请求，并对主体进行调度。在单处理器内，处理则是简单的信封类，而主体是信件类或为处理服务的类。这两个类是用指针联结的（如图 11.3 所示）。任何给定的主体实例完全受单处理器所控制，而主体在几个处理器上可以有使节（ambassador）类，它包含了一个主人的主体。即使当从多个处理器中引用时（或从多个名字空间引用时），主体（信件类）仍保持不变。在与主体相同的处理器上的处理类直接引用作为信件类的主体，这是在单处理器的情况。远程对象的访问总是通过处理类进行的，但该处理类用使节类（以替代主体类）转送请求。

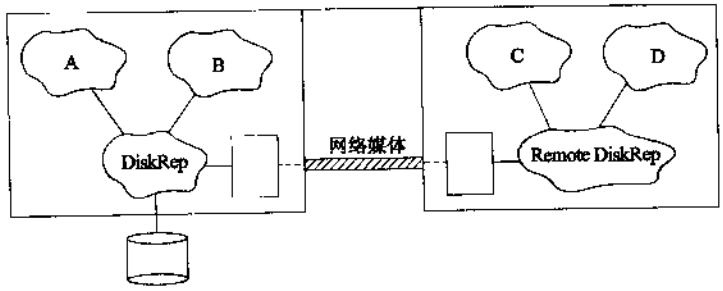


图 11.3 使节类

使节对象做的是把这些论据封装到消息中，并把它们透明地传送到外部名字空间中的“实”主

体类中的工作。例如，使节 RemoteDiskRep 封装 read 和 write 请求，把这些请求从处理类 C 和 D 发送到另一个处理器的 DiskRep。操作系统按照消息到达通信链路线调度，并发送消息到远程处理器，在发送终结后，调度启动另一个线程。操作系统在接受之后把消息发送到主体对象 DiskRep。该主体对象可以通过成员函数指定的目的来调度，再译码该消息，并在 DiskRep 上执行合适的操作。

注意：具有返回值的成员函数是不容易被分配的，因为跨越处理器边界调用这些成员函数要求在调用和返回之间实时中断。线程调度环境在诸如实时中断期间没有明确地准备存入返回地址和其他动态线程数据（自动变量）。线程的一个重要特征是大部分系统状态驻留在对象数据中，以替代在栈上的数据，以便能够请求错误分析和恢复。

如果返回值是必需的话，则它们可以作为回调（callback）来实现。当产生服务请求时，原始的服务请求可以指定回调的例行程序作为函数指针和功能元件（functor）。当主体对象 DiskRep 完成它的任务时，就能够发送消息返回到引用的处理器。该消息被调度到使节对象，能够引用回调。如果处理类对象通过在相同处理器（例如，从 A 中）的主体对象满足请求的话，那么主体能够直接引用回调完成该请求。如果主体无须实时中断就能够满足该请求，那么该回调作为原始请求在控制相同线程的基础上引用。如果该请求远程满足或者具有实时中断，则回调是在不同的线程上进行的。除了同步技术之外，访问者引用在 A，B，C，D 上的服务的透明度也是不同的。

例外处理（Exception Handling）

在一些要求特别关注的异常条件下可以产生的一个例外。例外的例子包括陷阱（诸如用零除个数）、中断（例如，信号数据由外围接口送达）和其他各式各样的软件错误（例如，企图引用不存在的虚拟函数）。例外处理是应用的一种特性，没有一种单一的方法能够适合所有的需求。本节将展示一些典型的例外处理方法，以便更好地理解和帮助设计容错软件。

在诸如编辑程序、编译程序、游戏程序以及其他程序中，可以充分地整理或“终结”资源，打印消息，然后出口。资源终结的例子包括关闭打开的文件，释放信号标杆，调整在共享资源上的引用计数，如此等等。通常，这样的清理自动地在对象析构函数内执行，所以如果所有的对象在例外处理都合理地被清除掉，则清理工作正好宣告结束。

在建立例外处理模型中的重要问题是：恢复单位是什么？在传统的微型计算机环境中（或者许多其他环境中），恢复的单位是进程，而例外产生的进程被取消或重新启动。因为进程放弃用施动者（Actors）或线程（Thread）作为更自然的对象调度规则，所以必须要寻找到新的恢复抽象规则。例外处理的一种模型是用函数（成员函数或全局函数或静态函数）作为恢复单位。这种例外处理机制已经纳入到 C++语言的国际标准之中。

在以函数为基础的例外处理中所存在的主要问题是当例外发生时，如何自动地清理（局部函数变量）。如果一个例外发生，而程序正在执行几个深度调用的函数，则对象就需要对所有在栈上激活的记录的函数进行清除。当前的执行函数应当给出例外恢复的机会；调用它的函数同样也应当在它所处的层次上得以恢复，如此等等，直到足够多的层次函数达到为止，其目的是使这种错误不被传播得更远。

在 C++例外处理模式中，给出容错处理的每个代码后段必须在 try 块中执行：

```
void foo() {
    try {
      // 在这里调用给定的容错处理
```

```
    }
// 这里调用没有给定容错的处理
// (除非 foo 从另外一个 try 块中
// 调用容错处理)
}
```

例外既可以在实时环境中（诸如用零除一个数）产生，也可以用 throw（抛弃）语句显式产生，这包含了一个被抛弃例外的类型的指定表达式。例外在 try 块中由代码产生，或者在该块内写入函数调用的代码，把控制转移到处理程序部分。例外处理程序部分用 catch（捕获）语句编码，当处理程序获得控制时，就执行处理和指定的语句（当然要对例外类型进行说明）。下面来看一个简单的例子：

```
void foo() {
    try {
        bar();
    }
    catch (const char *message) {
      // 获得一个指向 throw（抛弃）参数的指针
    }
    catch (const List <String> &message) {
      // 获得一个 throw 参数的临时拷贝
    }
}
void bar() throw(List <String> , const char*) {
    List < String > errorReport;
    ...
    throw errorReport;    // 引用第二个 catch 语句
    ...
    throw "Help!"         // 引用第一个 catch 语句
    ...
}
```

注意，作为接口一部分的 bar 的说明，例外将把它抛弃，在例外产生的点上没有执行恢复。

当例外发生时，这种机制对激活这些函数的局部对象的恢复做得很好，并且这种机制很适合包容器类、串和在通用库中找得到的其他公共类。C++的例外处理没有把终结对象的内存从堆栈中收回，它既没有处理异步的错误，也没有处理重叠式错误，这些情况都属于信号处理的领域。我们假定，这些对象在例外处理中用显式代码清除。

在实践中，复杂的容错系统需要更全面的恢复策略。首先，要能够处理与程序设计语言抽象关联不同的事件，诸如内存局部错误和辅助作用的设备失效。这样的错误是相对于“主”线程执行的异步，并且它们的恢复状况的发生与软件正在做些什么无关。这样的容错与发生在幕后的事情有关，而且能够将其分为子范型的恢复（低层中断，诸如内存局部失效）和超范型的恢复（在上下文、进程和处理器层上的恢复）。这种经验规则在连续运行的系统中恢复需要在技术的最低可访问层上访问。

第二，在 try/throw/catch 机制中恢复函数调用的嵌套（或与此等价的是相邻栈框架的集合）。在这些地方恢复的单位用对象来替代。例外处理例行程序服从高层错误处理的重新初始化工作。损坏的和毫无意义的对象可以通过在对象的适当位置或采用其他合适的技术引用它的构造函数来复原。

第三，系统层观点需要处理系统层错误，诸如内存耗尽。系统恢复模式至少可以删除关键性资

源。C++的例外处理可以认为是检测点技术，它是短期的（例如，如果使用线程调度技术，可使一个线程复原）和局部的（仅仅是局部数据“恢复”，若需要恢复全局资源就必须涉及到上下文）。不同的恢复方法适合不同的系统调度，程序设计语言不能够提供单一的、全面令人满意的解。

最后，一些错误在发生时可能捕获不到。如果一个错误在一些远程相关数据上通过第二次，第三次和第四次被有效地检测到的话，则例外处理例行程序对该问题的修复将不具备上下文关系。修复必须把软件逐步提升到更全局的观点，例外处理是一种提供逐步提升的机制，它不仅是这样一种机制，而且必须把连续运行系统中有效的全局恢复策略耦合起来。

目前，大多数 C++环境都支持例外处理技术。代理技术已经被开发成例外处理所推荐的预先实现。

可靠性的助手——垃圾收集

未引用内存的垃圾收集（garbage collection）与未引用对象的自动终结（引用析构函数）的耦合成为重要的恢复部分。在第 9 章我们已经描述过这种技术的习惯用法。如果一个对象被检测为无用，则应当使用它自己的析构函数释放所有已分配过的资源。通常，最好把这样的内存返回到空闲池中；实际上，引用它们的析构函数可能在从未定义的指针中启动了大量的错误。于是，就遗留下了有效的、现存的未引用对象，这正是垃圾收集技术要解决的问题。子范型的错误处理，当这些错误被激活时是不能够“做正确的面向对象处理的”，因此释放无用对象也是如此。作为一种模拟，在分时系统上的一个进程，如果认为用零除一个数的结果处于异常终结状态，则可能不能关闭所有打开的文件。

垃圾收集有 3 条可以帮助恢复的工作：

（1）它能释放与未引用对象有关的内存；

（2）如果垃圾收集引用了未引用对象的析构函数，则除了内存（文件描述字、虚拟内存寄存器、信号标杆等）之外的资源也被释放；

（3）如果析构函数用垃圾收集器引用，那么，仅引用未引用的对象被释放，如此等等，可能会潜在地损坏整个资源树。

垃圾收集是需要付出一定代价的技术，并且有时要求编写繁琐的代码。但是，它能产生适合应用的 C++代码，并仍能用于系统手工编制的代码部分，以满足系统的可靠性需求。

作为恢复单位的上下文

当使用线程时，就不再有进程的概念了。一个进程不能够取消和重新启动恢复手段。但是，上下文管理它们自己的资源，并能够作为恢复单位处理。服务器对象能够请求重新初始化所有局部上下文对象，也许还能处理共享的上下文对象。

复原对象

即使在最好的设计恢复模式中，意外仍会发生。如果试图要访问一个已经消失的对象，那将会发生什么情况呢？首先我们要避免发生这种事情，其次若触发这种错误的根源，分析起来是很困难的。其中一种解决方法是用复原对象（Zombie object）替代被删除的对象，该对象具有控制、故障无碍的行为（可靠性行为）。

复原对象依赖于 C++使用的环境。大多数复原对象应当用虚拟函数表指针来构造。该对象包含

的就是这样一组指针，并且所有指针指向相同的虚拟函数表。此表至少分配了像系统中最大虚拟函数表那么大的存储区，并且填满指向恢复函数的所有指针。这样的复原对象可以出现在任何对象所出现的地方，并且开辟所有它的虚拟函数调用恢复函数的工作区。

复原对象能够用系统的 operator delete 或用重载的 operator delete 成员函数重新占有新近被删除的对象的存储区。

复原函数引用虚拟函数表（全部是被动故障保障操作），产生例外（让访问者知道它是间接引用的悬挂指针）或者在恢复系统中安排使之相信当前的对象是毫无意义的。然后挑选一种系统恢复策略的函数来处理。

一般来讲，几乎不可能阻止非虚拟函数引用悬挂指针。

练　习

11.1 静态系统设计与动态系统设计有什么不同？它们各自包含哪些模块，其具体作用表现在什么地方？

11.2 名字空间（namespace）的设计有何作用，为什么？

11.3 例外处理（exception handling）的设计有何作用，为什么？

11.4 对象的成员函数是如何进行线程调度的？

11.5 垃圾收集机制在辅助恢复系统资源中有什么特殊的作用？试阐述之。

附录 A

C++环境中的 C

与 C 完全兼容是 C++的最大优势，因此可以在不放弃原有的 C 程序的基础上直接转向面向对象的程序设计。C++与 C 的这种兼容性，使得人们在保留现有的 C 应用程序的同时，又可以在 C++的基础上进行新的软件开发。因为 C++具有和其他语言模式混合进行编程的特点，这当然包括 C，所以我们可以在同一程序中同时使用这两种语言。作为 C 的超集，C++在技术上和 C 完全是兼容的，但在概念上是和 C 完全不同的语言。仅仅会使用 C++编译器是不够的，还需要掌握 C++的思维方式、C++的设计方法和 C++的习惯用法。附录 A 在诸多方面将介绍在 C++环境中 C++与 C 的不同之处。

A.1 函数调用

C++引入了一种在 C 语言中不能涉及到的范围层次，这个范围层次就是类（class）。类可以用抽象数据类型（abstract data types）或用户定义类型来构造。在 C 中支持局部作用域和全局作用域；在 C++中支持 3 种形式的作用域：局部域、全局域和类域。

在 C 中，变量可以出现在多层嵌套的函数内（用大括号表示），而在一个结构（struct）内的名字，可以在局部域和全局域内完全相同。但在 C 中，所有函数，既是全局的又是静态的（在源文件中的全局），这是不合法的。

在 C++中，函数可以说明在类中，这样的函数称作类中所包含的成员函数。这就是说，如果有一个全局函数 foo，一个类 F 包含了函数 foo，这两者是截然不同和毫不相干的函数。我们先记作

```
F::foo
```

即全局函数一个是::foo，或者有时就是 foo。

如果代码在类 F 内，我们可写成

```
foo ();
```

有一点要提醒读者：我们不能很明确地告诉程序员是否要调用在类 F 中的 foo，或者要调用全局的 foo。读者必须通过非寻常的工作，通过类，最后才能决定调用的全局作用域在何处。为了帮助阅读和理解这些代码，我们采用 C++程序中方便而清楚的使用类的形式，这个方便的记号是全局作用域的函数，即使用全局域的标识符::，亦即

```
::foo ();
```

这是一个记号，它表示在代码产生时不改变（对这个例子），并在编译时不会产生二义性。

A.2 函数参数表

C++要求在说明函数参数表与传统 C 在风格上只有一点细微的不同，而 PASCAL 程序员将发现 C++与 PASCAL 的风格是类似的：

C 代码：

```
int main (argc, argv)
int argc;
char *argv[];
{
   ...
}
int func ()
{
   ...
}
```

C++代码：

```
int main  (int argc, char *argv[])
   {
     ...
   }
   int func()
   {
     ...
   }
```

从指定的接口到函数的简单语法有两个目的：一是在函数定义中函数体作为头；另一个是它是作为一个函数接口的规格说明（称作函数原型）来使用的，必须出现在函数调用之前。

A.3 函数原型

一个好的 C 程序一开始就应有函数说明文件。某些程序员把这些说明放在每个函数内部，表示在内部使用；另一些程序员把说明放在文件头部；再有一些把所有的说明放在头文件中，用#include 放在头文件上。

在 C++中，最好的组织形式是把说明放在头文件中。如果需要某些函数和类，则通常的做法是将这些函数和类的说明提供在头文件中，用.h 后缀驻留在指定的头文件目录上。如果在程序中用#include 头文件，那就要对程序做类型检测存取。在头文件中进行说明，不仅确保了类型的安全，而且避免了在需要说明的地方老是重复说明这个函数和类。

在大多数 C++环境中，所有系统函数都在头文件中说明，例如，在 UNIX 操作系统安装中，C++环境的头文件包含了核和库函数的说明。为了得到这些函数的说明，必须正确地使用#include 头文件。

在传统的 C（Kernighan and Ritchie）中，在使用该函数之前没有说明，编译器通常不发布警告信息而只是粗糙地产生目标代码，并且编译器假设所有未说明的函数均返回整数。

另一方面，C++没有上述约定，只是强制性地进行显式说明：所有标识符和函数必须在使用前加以说明。下面的例子就是对传统 C 与 C++之间的比较：

C 代码：

```
extern double myfunc ();
extern double cos();
```

C++代码：

```
extern double myfunc (int, char);
extern "C" double cos(double);
```

在上面的例子中，ANSI C 与 C++之间，除了 ANSI C 中没有 extern"C"连接之外，其他规则都是相同的。

注意，这个连接（extern "C"的意思是指明 C 语言），表示程序的语言可以是说明的显式部分。

一般地说，一个直接的连接对所有符号（函数和数据）是需要的，它能被其他程序设计语言存取，在大部分实现中，指定C连接对数据没有影响，而对C++函数在目标代码中与C连接的目标代码中有不同的名字。如果显式连接没有指定，则假定为C++连接（如myfunc）。

函数原型指的是函数的参数类型和返回类型。编译器用函数原型至少可以保证在调用该函数时，可以提供正确的入口参数和类型。C++要求在调用函数时必须严格按照函数原型，其中包括参数类型。若事先没有告诉编译器函数原型就想简单地去调用，这样做是行不通的。下面例子说明C++能自动地将整数转换到浮点数：

C代码：

```
extern double sin();
sin(1);  /* 未定义 */
sin((double)1);
sin(2.0);
sin(3.0);
```

C++代码：

```
extern double sin (double);

sin(1);  // 自动地转换成 double
sin(2);  // 自动地转换成 double
sin(3.0);
```

A.4 调用引用参数

在C（或者C++）中，参数传递的通常语义是把被调用函数的每个参数进行复制，这就是传值方式。在传值方式下，函数从不引用实际参数，它所处理的仅仅是实际参数的一个备份，这些备份存放在堆栈中。因此，这些局部备份的变化并不导致实际参数的变化。当函数执行结束时，该函数的参数区在堆栈中被激活，由栈中自动弹出，于是这些局部备份值便丢失掉了，即在函数返回时，复制的值就丢失了。

在传值方式下，实际参数的值并不发生变化，即程序员在进行函数调用时不必保存参数值，也没有形成对实际参数潜在的被函数改动的可能，从这点来看，传值是参数传递过程中一种合理的隐含机制。但是，传值并不适用于所有函数参数：（1）当需要将一个类的较大的对象作为参数传给函数时，用于存放该对象备份的空间和存放过程所需的时间都比较大，这在实际应用时是不能容忍的；（2）当参数值必须被修改后返回调用者时，传值也不再适用。下面例子中swap()函数就需要实际参数作改变，但在传值情况下都是办不到的。

```
void swap(int v1, int v2)
{
    int temp = v2;
      v2  v1;
      v1 = temp;
}
```

这里，swap()只交换了备份值，而实际参数并没有任何变化。下面是调用swap()函数的例子：

```
#include < stream.h >
void swap (int, int);
main()
{
  int i = 10;
  int j = 20;
  cout << "Before swap(): \t i:"
```

```
        << i << "\tj:" << j << "\n";
    swap(i, j);
    cout << "After swap (): \ti:"
        << i << "\tj:" << j << "\n";
}
```

编译并执行后，得到下列结果：

```
Before swap():i:10 j:20
After  swap():i:10 j:20
```

在 C 语言中，有时将变量地址作为参数传给函数（当然，函数参数带有“*”号），它提供给调用者能够改变变量的值。在 C++中，提供给程序员两种替代传值的参数传递方法。第一种方法：将形式参数声明为指针，因此，上例中的 swap()可改写成 pswap()：

```
void pswap (int *v1, int *v2)
{
  int temp = *v2;
    *v2 = *v1;
    *v1 = temp;
}
```

当然，main()函数也要修改成调用 pswap()，并改动声明，使 pswap()的原型出现在它的调用之前。现在，程序员必须传递这两个参数的地址，而不是它们的对象值，这同 C 程序是一样的：

```
pswap(&i, &j);
```

编译并执行后，给出了所期望的结果：

```
Before swap():i:10 j:20
After  swap():i:20 j:10
```

第二种方法：在 C++中，避免了指针的需要，并且不引用显式指针，即将形式参数声明为引用型。我们将 swap()函数改动为 rswap()：

```
void rswap  (int &v1, int &v2)
{
    int temp = v2;
      v2 = v1;
      v1 = temp;
}
```

在 main()中对 rswap()的调用与原先对 swap()的调用形式是一样的：

```
rswap(i, j);
```

编译并执行后会发现 *i* 和 *j* 的值已被正确地交换了。

调用引用参数是将实际参数作为左值传给函数，这样会产生两点效果：

（1）在函数内对参数值的修改不再作用于局部复制，而是针对实际参数进行的。这就是为什么 rswap()可以正常工作的原因。

（2）在传递大型的类对象时，也不再有高额的空间与时间开销。它的传递方式类似于指针，不像传值那样复制整个类对象。

考虑下面 incr 函数，当增加自变量时，同时返回旧的自变量的值：

C代码：

```
int incr (i)
int  *i;
{
  return (*i) + + ;
}
int main()
{
  int i, j = 5;
  i = incr(&j);
  /* i = 5, j = 6 */
  ...
}
```

C++代码：

```
int incr (int &i)
{
   return i + + ;
}

int main()
{
  int i, j = 5;
  i = incr(j);
  /* i = 5, j = 6 */
  ...
}
```

其中 int &i 在C++语法中，意思是向变量 *i* 传递一个引用。函数形式参数（*i*）变成有相同值（或相同对象）的另一个名字，作为在 main(*j*)实际参数进行传递。*i* 和 *j* 都是同一个对象的名字。一般来说，此种引用称为引用性调用。通常，引用变量总是可以建立已经命名的对象的名字：

```
int m = 5;
int &n = m;  // n 是 m，而 m 也是 n
  n=6;       // 现在 m 和 n 都是 6
```

引用的含义是使用更多的引用性调用，作为隐含状态，函数的返回值也是通过传值来完成的。当需要返回一个大型的类对象时，指针和引用这两种形式是非常有效的。在此种情况下，对象本身并没有被复制。

函数返回值为引用型的第二个用法是使这个返回值作为赋值号左侧的元素。因为返回值是引用型的，所以实际上返回的是对象左值。这一点对于像 IntArray 类中的下标操作符函数来说是非常重要的，因为它必须赋予返回值的读和写这两种能力：

```
int & IntArray :: operator[](int index)
{
   return ia[index];
}
IntArray myArray(8);
myArray[2] = myArray [1] + myArray[0];
```

A.5 参数个数不定

在程序的函数之间可以通过两种方法进行通信。一种是使用外部变量；另一种是使用函数的参数表。参数表提供了另一种用于函数之间通信的方法。参数表和函数的返回类型一起定义了函数的公共界面部分。只要函数的公共界面不变，即使函数的实现部分发生变化也不影响调用它的程序，由此提高了程序的可维护性。

调用时参数个数不够，或使用了错误的参数类型，都是在 ANSI C 中常见的，也是较为严重的运行错误。现在，C++中引入了强类型检查机制，使这些错误可以在编译时被查出。下面，我们以在 C 中 printf 函数的例子，来看看在参数个数不定时是如何说明、调用和实现 printf 函数的。

既不知道自变量个数，又不知道自变量类型的函数是这样说明的：

```
extern "C" int printf(...);
```

在圆括号内的省略号“…”的意思是说：参数的类型和个数是不定的，在编译时是不知道的。带有这种形式的类型系统使调用者能够传递任何类型的任何个数的自变量。这种不定的形式让用户在初始化参数时说明参数的类型，并确定参数的个数。下面是另一种形式：

```
extern "C" int printf(const char * ,...);
```

这样的说明是典型的由应用者用#include 在头文件中说明，头文件由 C 库和操作系统本身提供，而且在大多数 C++编译器中也是提供的。

在 C++中调用一个函数参数个数不定，就好像在 C 中调用一样：

```
#include <stdio.h>
// stdio.h 说明 printf
int main()
{
printf ("hello world \n");
int i = 1, j = 1;
printf("% d plus % d equals % d \n", i, i + j);
return 0;
}
```

定义一个不定参数个数和类型的函数在 C 中要求更多，采用 varargs.h 的 stdarg.h 机制，允许用可移植方式，用代码写上参数的不定个数。C++与传统 C 不同，至少一个形式参数在函数说明部分要加以说明。这个参数的说明仅仅保证从函数调用体内用可移植方式存取实际参数：

```
# include < stdarg.h >
# include < stdio.h >
int printf (const char *pl, ...)
{
  va_list ap;
  va_start (ap, pl);
  ...
  switch (fmt)
  {
    case'f':
      double d = va_arg(ap, double);
      ...
    case's':
      char *string = va_arg(ap, char);
      ...
    case'd':case'c':
      int val = va_arg(ap, int);
      ...
    }
  va_end (ap);        // 返回之前调用
}
```

这里，C++的标准自变量程序包的使用与传统 C 的某些程序设计实践是不同的，传统 C 是用栈布局的策略来实现的。在 C++中，可通过使用 stdarg.h 的机制，可以达到确保实际自变量传递到不确定自变量个数的子程序中。

为了增强代码的可读性和可维护性，一个好的程序设计实践者劝告大家：应使用参数个数不定这种特殊的参数形式。正如上例所说的，使用 stdargs 程序包机制，提供了程序设计上的方便，并提高了可移植性。圆括号内省略符“...”说明比传统 C 提供了更加方便的记号。stdargs 可以经常缓解函数的重载和自变量的缺省，增强了类型的安全性和使用上的便利。

A.6 函数指针

函数指针的工作，在 C++中几乎与 C 中的做法是相同的。不同之处在于自变量类型的信息必须要提供：

C 代码：

```
void error(p) char *p;
{
  ...
}
void (*efct) ();
int main()
{
    efct = &error;
    (*efct) ("error");
    efct ("also calls error");
    return 0;
    }
```

C++代码：

```
void error (const char *p)
{
  ...
}
void(*efct) (const char* );
int main()
{
  efct = &error;
  (*efct) ("error");
  efct("also calls error");
  return 0;
}
```

另一个例子是大家熟悉的 signal 系统调用。signal 说明作为一个函数有两个参数：第一个是 int；第二个是 SIG_PF（在头文件中提供的 typedef 类型）。该函数本身返回一个指向函数的整数参数：

C 代码：

```
int efct2();
typedef int (*SIG_PF) ();
SIG_PF fptr;
extern (*signal()) ();
fptr = signal (3, &efct2);
```

C++代码：

```
int efct2(int);
typedef int(*SIG_PF) (int);
SIG_PF fptr;
extern( *signal(int, SIG_PF)) (int);
fptr = signal(3, &efct2);
```

下面的例子类似于上面的例子，但却具有更为特殊的自变量类型：

C 代码：

```
extern(*f()) ();
( *f(1, 'a', 123)) (1.0, 2);
( *(f(1, 'a', 123))) (1.0, 2);
f(1, 'a', 123) (1.0, 2);
```

C++代码：

```
extern int ( *f(int, char, long)) (double, int);
( *f(1, 'a',123)) (1.0, 2);
( *(f(1, 'a',123))) (1.0, 2);
f(1, 'a', 123) (1.0, 2);
```

函数指针说明，经常用 typedef 来处置。这里 signal 例子的重演使用更多的 typedef：

C 代码：

```
typedef int ( *SIG_PF) ();
extern (*signal()) ();
```

C++代码：

```
typedef int (*SIG_PF) (int);
extern SIG_PF signal (int, SIG_PF);
```

下面是使用 typedef 更为突出的例子：

C 代码：

```
typedef int (*FP) ();
  extern FP (*f) ();
f(1, 'a', 123) (1.0, 2);
```

C++代码：

```
typedef int (*FP) (double, int);
extern FP f(int, char, long);
f(1, 'a', 123) (1.0, 2);
```

A.7 const 类型说明符

在 C++中，使用 const 类型说明符可以将定义的一个“变量”说明成常量。使用 const 可以替代用#define 定义所声明的常量。const 不但说明了数据，而且申明它是常量，即它的值不能改变。const 类型说明符与#define 常量相比具有更多的优点：首先，分别独立编译的模块可以共享一个 const 数据拷贝，这就避免了以下众所周知的错误的发生，即#define 的值已经改变了，但所有的程序模块并没有被重新编译；const 的另一个优点是它创建了一个真正的数据，这有时使我们追踪起来更加方便。const 还可以用在结构、类和数组中，这是#define 所做不到的。

一个 const 说明符允许编译器强制检测指定的指针（它指向什么），或者强制检测在初始化后写入一个变量。我们来看下面 3 个例子：

例 A.1：用 const 替代#define。

C 代码：

```
#define CTRL_X '\30'
#define msg "an error"
```

C++代码：

```
const char CTRL_X = '\30';
const char *msg = "an error";
```

两者共同的：

```
int main()
{
  char c;
  ...
if(c = = CTRL_X) {...}
printf("% s \n", msg);
  return 0;
}
```

C++编译器能够确保 const 创建的是一个真正的数据，用户不能改变常数的值。如果基于某种目的要求改变常数的值，则一定要用显式强制作法，但为了保险起见，还是不要这么做。

例 A.2：常数指针。

指针已经被称为“数据的转移”。但是，在 C 和 C++程序中，指针有一个重要的规则：作为有效的在大的数据记录或者数组函数之间传递的一种服务方式。程序员有时将指针作为“只读方式”传递到那个数据中。

const 保留字可以通过直接方式或者指针方式控制修改（到）那个数据。我们来分析第一个例子：

```
char *const a = "example 1";
```

这里，我们说明 *a* 是一个指针，它是一个常数，即 *a* 的内容不能被更改。但“*a* 的内容”是什么呢？实际上，它是某个存储块的地址。所以这个例子的说明告诉我们不能更改一个指针的位置：

```
a = "example 2";   // 在编译时的非法标志
a[8] = '2';        // 成功地创建了“example 2”
```

能够合法地出现在赋值左边的一个表达式称为左值的（l_value）。这里，*a* 是不合法的左值，而 *a*[8]则是合法的左边的值。

为了防止一个指针使用左值来修改这个数据，我们需更改说明，使指针指向一个常量本身。下面有关 *b* 的说明，使字符串“example 2”的内容不能更改，而 *c* 的说明防止指针本身被修改，符号 *c* 可以作为不受影响的常量。

```
const char *b = "example 2";
const char *const c = "example 3";
```

虽然 *a* 不是左值的，而*a 是左值。*b* 指针是左值，但*b 不是。*c* 和*c 都不是左值。注意，const 类型说明与下面的说明符有直接的联系：

```
 a = "ex4";  // 不可以，单说明 char *const
*a = 'E';    // 可以，a 是左值，说明为*
 b = "ex4";  // 可以，b 是左值，说明为 char*
*b = 'E';    // 不可以，b 被说明为 const*
 c = "ex4";  // 不可以，c 被说明为 char *const
*c = 'E';    // 不可以，c 被说明为 const*
```

例 A.3：说明有 const 自变量的函数。

为了拓宽一个对象是常量，形式函数参数的说明必须指定 const。如果试图传递一个 const 实际参数到期望为非 const 的形式参数的函数，编译器就会发布一个错误信息。当函数说明结束时，编译器确保 const 的常量性质不变（无论在程序中经过多少次传递）。

考虑下述说明：

```
extern char *strcpy (char* , const char*);
```

如果我们有如下说明：

```
char a[20];
const char *b = "error message";
char *c = "user prompt";
```

则 C++允许

```
::strcpy(a, b);        // 把 const 拷贝到非 const 内存中
::strcpy(a, c);        // 可以把 const char*自变量
                       // 传送到非 const 中去
::strcpy(c, b);        // 像:strcpy(a, b) 一样
```

但不允许

```
::strcpy(b,c);         // 不可以把非 const 自变量
                       // 传送到 const char*中去
```

当然，我们可以使强制检测机制无效：

```
::strcpy((char *)b, c); //okay now
```

但是为什么我们要把 b 说明成 const char*呢？

const 保留字在传统的规则描述中扮演着重要角色。

A.8 C 代码的接口

我们知道，在许多成功的计算机应用系统中都包含了系统所共享的低层代码，或者是在一个庞大的计算机项目中遗留下来的共享代码。由于历史或者社会因素，在新的计算机应用系统的开发过程中，可能需要将 C 和 C++混合起来使用。基于这些原因，C++语言已经提供了与 C 语言的接口。这一节我们从 C++各个侧面讨论（或寻找）这些外部连接问题。首先我们来讨论设计的考虑以及有关语法和程序管理方面的问题。

A.8.1 设计的考虑

在兼有 C 和 C++的程序中，它本身的内涵已经超出了语法和说明（共享）以及对这个程序连接的机制。C，C++和其他语言必须对这个程序的许多方面作出评价，比如，程序安排的合理性、在程序解中所扮演的角色以及使用项目的工具对程序的兼容性等等。

使用 C++的一个主要原因是它所具有的能力，此能力的重点在于用抽象数据类型和对象的设计实现。C 代码几乎很少能反映出程序设计的风格，它通常用功能分解设计方法，或者从自动代码生成工具中输出，诸如语法生成器或者 CASE 工具。所以，C 和 C++代码的混合至少反映出是两种不同设计风格的混合。

这样的混合频繁地在软件产品构造历史或组成上发生。C++既能够用面向对象设计，又可以用面向过程设计来写新的程序代码。我们对已有的成熟的代码，不需要重写（重写本身就是一种浪费）；或者我们用预先产生的 C 代码生成器（如 YACC）来写，然后将两种代码混合在一起组成新的应用程序。当然，何时或者怎样对两种不同代码进行混合，完全由设计者决定。

下面列举两个典型的 C 和 C++代码进行混合的例子：

在 C 环境下的对象库

考虑一个用 C++语言写的具有数据库管理的一个大型的 C++应用程序。另一个长而标准化的 C 语言应用程序，需要嵌入到数据库的通信之中。在数据库管理中，C 语言程序要引用对象的 C++成员函数，而且 C 和 C++软件需要共享数据结构，所以，在 C 程序的“main”部分要使用 C++对象作为服务于对象的一个调用库。

在 C++环境下的 C 库

我们已有用 C 语言写的，不论是在公共领域，还是在私有领域中，都具有的丰富的软件。C++程序能够建立具有一定效率的软件，要求能够调用 C 函数，而且能够存取 C 的数据结构。这样我们就能够在 C 和 C++之间很容易地设计两者的接口。一个例子是将 C++程序写在已有的图形库的头上，而新的 C++代码使用操作系统的接口。

A.8.2 与 C 语言连接

C++提供了名字重载。这就是说，两个函数可以有相同的名字。从习惯上来看，对于在不同类型的对象上完成类似任务的函数，让这些函数有相同的名字会更好些。但要基于提供的不同的参数类型，使之在上下文调用中不可以有模棱两可的选择。对一个简单函数名的多重说明可以出现在程序的头文件中，每个文件具有不同的参数表。它提供了返回型的类型。在 C++源文件中，一个独立的函数体对每个这样的说明都已提供了。

在使用 C++版本中，可能遇到这样的麻烦，即依赖于文件头#include 行排序的是什么次序。例如，cookie.h 和 cake.h 都包含在函数 print 的说明中：

```
cookie.h:    extern void print (cookie *);
cake.h:      extern void priht (cake  *);
```

程序员这样使用这些函数：

```
overload print; // C++重载构造
# include "cookie.h"
# include "cake.h"


int f1()
{
   cookie c;
   cake d;
   print (&c);    // 调用打印内部名的函数
   print (&d);    // 调用有编码名的函数
}
```

第一个 print 的内部名是 print，而第二个 print 是一个区别于第一个的内部表示的后缀。这样做是无可非议的，除非系统的另外一部分代码为如下形式：

```
overload print;
#include"cake.h"
#include"cookie.h"


int f2()
{
   cookie c;
   cake d;
   print(&c); // 调用有编码名的函数
   print(&d); // 调用打印内部名的函数
}
```

这两个#include 的排序直接影响到函数调用的次序。一个可能的情形是要求所有重载的说明置于单一的#include 文件中，另一个可能的情形是对相同名字的处理，像 print 语句一样重复发生在许多库中，这样做是没有任何理由的。

当前，大多数 C++编译器，对每个不同的函数生成一个惟一的内部名，将函数自变量类型的后缀增加到函数的原始名中。在 C++中，overload 保留字现在已退化了。但这些却触发了第二个问题：一个调用 read（读）到 read_something 的一个内部名（其中“_something”是导出了函数自变量的类型）。如果函数 read 本身是用 C++写的，而所有期望的调用 read(2)函数是用 C 写的，那么程序将不能连接。

为了解决这个问题，如果上面所列举的情况不用 C++写的话，函数就必须使用一种外来语言。如果我们希望调用 read 的 C 版本，我们必须这样写：

```
extern "c" int read (int, char*, int);
```

这样，我们还可以照样说明若干个是 C 的函数：

```
extern "C"
{
  int read (int, char*, int);
  int open (char*, int, int = 0);
  int write (int, char*, int);
}
```

这些说明不可以用在一个类或函数的内部，它们必须在最外面的层次上，最好放在头文件中，用户也希望放在#include 文件中（取得子程序的说明）。

某些 ANSI C 语言的头文件，对 C++需求有着精确的定义（除了用 extern"C"构造来定义函数连接之外）。一个 C++程序希望用这些头文件之一就能简单地把#include 直接在内部连接：

```
extern "C"
{
#include<C funcs.h>
};
```

如果该头文件除了函数说明之外（如结构或常数说明），没有别的构造的话，这是不重要的，这些处理是理所当然的。例如，在 extern "C"范围中，上述这个函数说明对类成员函数的说明没有任何影响。

A.8.3 从 C 调用 C++

一个 C++类成员函数不应直接调用 C 代码，而应通过全局 C++函数作中介完成这样的调用。一个不利的实践采用这样的做法：把在 C++系统层上的 C++名字映像成 C 名字，而在原始层次上直接调用了“被篡改”的名字：

```
struct stack
{
... // 模仿 C 的说明
... // C++类栈结构
};
struct stack s;
...
push_5 stack Fi_v(&s, 10); /* 不推荐作法 */
```

这样建立的代码，一是不可移植；二是对人多数应用者来说要求更多的关于编译器内部构造的知识；三是不能保证将来升级的编译器，甚至程序员自己本身所使用的编译器的工作是否正确；四是理解上很困难。

一个较好的方法是写一个 C 语言接口函数，嵌入到 C++编辑的模块中，这些 C 语言接口子程序可以调用通常的 C 函数：

```
C++代码：
class stack
{
   ...
};
extern "C" void CStackPush (stack *s, int v) {s ->push(v);}
```

这些接口函数能够作为“防火墙”来使用，在两种不同风格的程序设计语言中架起了一座桥梁。需要注意的是，此种设计一定要细致小心，但大多数还是成功的（接口函数在成功之中，也是不显眼的）。

A.8.4 在 C 和 C++之间共享头文件

用在头文件进行函数说明，对函数自变量和返回类型提供了一个单一的参考点，这种函数说明在程序中扮演着最重要的角色。如果对用户和提供者都用同样的头文件#include 函数，那么类型的安全是有保证的，甚至对重载函数的类型安全也是有保证的。我们希望这些工作对 C 函数也是适用的，即它们既可以被其他 C 函数调用，也可以被 C++代码调用。而 ANSI C 说明是，

```
extern char *foo (short, long);
extern void bar (char, const char *);
#define SIZE 10
extern char *text [SIZE];
```

不同于传统 C 语法，

```
extern char *foo();
extern void bar();
#define SIZE 10
extern char *text [SIZE];
```

而 C++的不同之处是，

```
extern "C" char *foo(short, long);
extern "C" void bar (char, const char *);
const unsigned char SIZE = 10;
extern chat *text[SIZE];
```

对 C 和 C++混合在一起的程序而言，怎样才能构造一个单一的共同的头文件提供给它们使用？有时，我们可以对所有 3 种语言使用一种具有共同标准的单一的语法。例如，对 SIZE 和 text 的 C++规格说明，可使用 C 风格。但是，C++本身有它自己说明的风格的优点，提供最好排除故障信息或者其他更多的有效代码。我们采用条件编辑——提供预先处理宏#if，#ifdef 和其他相关式——在单一的头文件中融合不同语言的风格。

只有 ANSI C 编译器预先定义了预处理定义_STDC_；只有 C++编译器定义了_cplusplus。在只有 ANSI C 和 C++的环境中，我们可以采用下面这样的方法：

```
#if _cplusplus
extern "C"{
#endif
   extern char *foo (short, long);
   extern void bar(char, const char *);
#if _STDC_
#define SIZE 10
#endif
#if _cplusplus
const unsigned char SIZE = 10;
#endif
extern char *text[SIZE];
```

```
    ...
#if _cplusplus
}
#endif
```

这似乎有点不太舒服：常数 10 出现在两个地方，所以，可能会在程序的执行过程中失去同步。下面这种方法可能会更好些，它采用修改 SIZE 所表示的值的方法，只需要在一个地方进行修改：

```
#if _cplusplus
extern "C"{
#endif
    extern char *foo (short, long);
    extern void bar (char, const char *);
# define COMMON_SIZE 10
#if _STDC_
#define SIZE COMMON_SIZE
#endif
#if _cplusplus
     const unsigned char SIZE=COMMON _SIZE;
#endif
char *text[SIZE];
    ...
#if _cplusplus
}
#endif
```

要 text 尽可能多地共享，需要保持修改协调的一致。

另一个经常遇到的混合的环境是在 C++中使用传统 C：

```
if _cplusplus
extern "C"{
#endif
    extern char *foo(
#       if_cplusplus
            short, long
#       endif
    );
    extern void bar (
#       if _cplusplus
             char, const char*
#       endif
    );
#define COMMON_SIZE 10
# _cplusplus
    const unsigned char SIZE=COMMON_SIZE;
#else
#  define SIZE COMMON_SIZE
#endif
extern char *text[SIZE];
  ...
#if cplusplus
```

```
}
#endif
```

如果你希望融合所有 3 种语言，在文件一开始就应有一个复杂的表示：

```
# if _cplusplus
extern "C" {
# endif
   extern char *foo(
#       if _cplusplus || _STDC_
            short, long
#       endif
   );
   extern void bar(
#      if _cplusplus || _STDC_
           char, const char*
#      endif
   );

#define !_cplusplus
#  define SIZE COMMON_SIZE
#else
   const unsigned char SIZE=COMMON_SIZE;
#endif
extern char *text[SIZE];
   ...
#if_cplusplus
}
#endif
```

一般地说，代码的读比写要多得多，大概是 10：1 的关系。我们对每种语言可以使用单一的#ifdef 交替进行，并对每种可读性结构进行复制，以期减少维护的冗余问题。

存有单一的共同的语法的地方，能够表达一种等价于 ANSI C 和 C++的说明。例如，有一种公共的形式可以使用 C++和 ANSI C 去说明无参数的函数：

```
return-type function(void);
```

上述这种形式在 ANSI C 程序中很清楚地表达为没有参数；如果 void 不是在这里，ANSI 编译器将接受具有任何类型任何个数的参数。在 C++中，一旦选用 void，它就是合法的，且有与 ANSI C 相同的语义。另一个对这两种语言的共同形式是能够用符号常数说明：

```
#if _cplusplus
extern "C"{
#endif
static const type symbol 1 - name=initializer;
extern const type symbol 2 - name;
#if _cplusplus
}
#endif
```

对这两种语言而言，很明显地 static 和 extern 是关键字。

这些通常的规则，对每个应用程序或管理程序的环境来说，应该广泛地加以应用，有一个公共的表示结构，对应用方面的更改将发现具有极大的方便。

A.8.5 将 C 数据输入到 C++

我们已经讨论了如何在 C 和 C++之间进行函数的存取，那么数据将如何进行呢？如果有一个 C 结构，要从 C++中存取数据，那么可以用相同的头文件，在两种语言之间说明该结构的布局。如果提供的一个结构说明是依附于 C 语法的，则此结构的“对象”能够在单一程序中毫无问题地共享 C 和 C++之间的代码。

A.8.6 将 C++数据输入到 C

上面已经讨论过，当我们写 C++代码时，可以使用已有的 C 数据格式，这里，我们来研究相反的工作情形。一般地说，我们不能增加 C 代码到已存在的 C++应用程序，而且期望 C 代码使用 C++数据格式。所以我们讨论相反的工作情形是很有必要的。

如要在 C++程序中发现该数据结构是无成员函数的单一 C 结构，那么可以作为 C 结构来处理，用 C 和 C++程序的类似的#include 描述就可以了。

这里有两种基本方法处理更复杂的情形。

（1）（不推荐的方法）

首先要确定 C++类的数据分布是怎样的，然后在一个 C 结构中复制它（C++）的结构。一个具有虚拟函数的类可以有意想不到的域，比如有哑成员，这时，必须提供一个 struct。这个领域的位置和长度依赖于实现。如果 C++环境允许从 C++产生 C 目标代码，那么可以从包含类说明的 C++头文件中自动地产生一个 C 头文件，通过编译器合适地选择和运行。但是，这样产生的一个头文件有一个不太舒服的域名，而且需要 C 程序员具备有关编译器的算法的某些知识才能工作。虽然手册中可以告诉我们使这两种接口保持同步的工作要领，但是错误的出现在所难免，因此我们不推荐此种作法。

例 A.4：用 C 代码存取 C++代码。

```
extern "C" const char char *EmployeeRecordName(void *);
extern "C" short EmployeeRecordYOS(void *);
extern "C"double EmployeeRecordSalary(void *);

class Employee Record {
friend const char *EmployeeRecordName(void *);
friend short EmployeeRecordYos(void *);
friend double EmployeeRecordSalary(void *);
public:
   EmployeeRecord(const char* , short, double);
   EmployeeRecord();
   void giveRaise(double);
   virtual void disbursepay();
private:
   char name [NAME_SIZE];
   short year()fservice;
   double monthySalary;
};
```

```
    extern "C" {
        void aCFunction (void *RecordPtr);
        const char *EmployeeRecordName (void *RecordPtr){
          return ((EmployeeRecord *) RecordPtr) ->name;
        }
        short EmployeeRecordYOS(void *RecordPtr){
          return((EmployeeRecord *)RecordPtr) ->years()fService;
        }
        double EmployeeRecordSalary (void *RecordPtr){
        return((EmployeeRecord *)RecordPtr) ->monthySalary;
        }
}
    ...
    Employee Record aRecord;
    aCFunction (& a Record);      // C应用子程序的调用
        ...                       // 给出C的一个指向
        ...                       // 某些C++数据的指针
```

（2）在 C++中提供了通过函数存取数据，为此采用某种“防火墙”策略，这已在上面有所描述。

在例 A.4 中列举了这样一个例子：某些 C 代码可以存取用 C++写的 EmployeeRecord 类的数据成员，C 代码给出了匿名的（void* 指针）作为参数来处理；它使用无引用的函数作为一个参数进行处理：

```
    extern char *EmployeeRecordName(void *);
    extern short EmployeeRecordYOS(void *);
    extern double EmployeeRecordSalary(void *);
    void aCFunction (void *RecordPtr){
        char *name = EmployeeRecordName(RecordPtr);
        short YOS = EmployeeRecordYOS(RecordPtr);
        double salary =EmployeeRecordSalary(RecordPtr);
        ...
    }
```

A.9 操作符引用返回值

一个引用变量既不是一个指针，也不是一个真实的对象或者真实的变量。更确切地说，它是一个对象的名字，即一个引用就给某些预先存在的对象起了一个新的名字。

对为什么 operator=或 operator[ ]返回引用值，这件事情的深层回答是一个允许返回值的引用，此返回值的作用类似于一个变量名。当然，一个变量名在任何地方都可以产生，一个具有相同类型的变量引用都是合法的。例如，一个名字可以作为左值（l_value）出现，但对一般的表达式却是不行的。如果 operator = 和 operator[ ]有返回引用值，就意味着这样的表达式可以作为左值出现。向量是一个特殊的重要的情况，我们要对向量进行赋值。

下面列举了简单引用调用的习惯的例子：

C 代码：

```
int incr(i)
int  *i;
```

C++代码：

```
int incr(int &i)
{
```

```
{                    =
   return( *i) + + ;
}
int main()
{                    =
  int j = 5;
  incr(&j);
}
```

```
return i + + ;
}

int main()
{
int j = 5;
incr(j);
}
```

这种习惯能够扩充函数返回值类型，还包含了这些操作符重载函数。事实上，许多建立的双目操作符应返回引用值。考虑下面这个例子：

C 代码：

```
struct  string {
   char *rep;
};

char index(t, i)
struct string *t;int i;
{
  return t ->rep[i];
}

int main()
{
  struct string x;char c;
  ...
  c = index(x, 5);
  index(x, 6) 'a';
  return 0;
}
```

C++代码：

```
1 class string{
2 public:
3     char operator[](int i){
4         return t[i];
5     }
6 private;
7    char *i;
8 };
9
10
11 int main()
12 {
13  string x;char c;
14  ...
15  c=x[5];
16  x[6] = 'a';
17  return 0;
18  }
```

这里，C 版本在编译时出现如下错误信息：

```
"t2.c", line 16:addressable expression required,
```

而 C++版本给出这样的错误信息：

```
"t2.c", line 16:error:assignment to generated function call (not an lvalue)
```

但是，如果我们对 C 版本增加一个间接的层次——在 C++版本中对使用引用返回值做一个模拟的修改——则所有的工作是成功的：

C 代码：

```
struct  string{
   char *rep;
};

char *index(t, i)
struct string *t;int i;
{
  return t ->rep+i;
```

C++代码：

```
class string {
public:
  char &operator[](int i){
       return t[i];
}
private:
  char *i;
};
```

```
}
int main()
{
  char x[80], c;
  c = *(index (x, 5));
   *(index(x, 6)) = 'a';
  return 0;
}
```

```
int main()
{
string x;char c;
c = x[5];
x[6] = 'a';
return 0;
}
```

练　习

A.1　编写 printf 函数简化版本，要求只用%d 和%s 格式处理。

A.2　编写有以下接口的 qsort 子程序：

```
void qsort (void *base, int nel,
            int elementSize, int( *compar) (void*, void *));
```

有以下假定：base 指向排序的基本表上的某一元素，nel 是该表的整个元素；elementSize 是单个表元素的大小；compar 是指向比较两个元素的函数的指针并根据第一参数是小于、等于或大于第二参数来判定是否返回小于零、等于零或大于零的整数。

A.3　编写我们所熟悉的 UNIX 函数 printf，malloc，read，write，pipe，open，close 和 signal 的说明。

附录 B

Shapes 程序的 C++代码

```
#include "Shapes.h"
/*-----------------------------------------------------------------------*/
/* File main.c --Shapes 例子的驱动程序
/*-----------------------------------------------------------------------*/
const int XSIZE = 800;
const int YSIZE = 1024;
void
rotateList (Shape *s, Angle a)
{
   for (Shape *f=s; f; f=f ->next())
     f ->rotate(a);
}
int
main()
{
   Coordinate origin (0, 0);
   /*
    * windowBorder 不仅是屏幕的框架，而且作为形状链表的表头。
    * /
   Rect windowBorder (origin, XSIZE, YSIZE);
   Shape *t = new Triangle (
     Coordinate(100, 100),
     Coordinate(200, 200),
     Coordinate(300, 100)

};
   windowBorder.append(t);

   Shape *c = new Circle(Coordinate(500, 652), 150);
   t ->insert(c);
   rotateList(&windowBorder,90.0);
   return 0;
}
#include <math.h>
#include <X11/X.h>
/*
 * File Shapes.h
 * 这个头文件包含了在演示形状库中使用的类的说明。
```

```
 * 许多成员函数在此处是作为内联函数来定义的。
 * /

 /*------------------------------------------------------------------------ */
 /* Angle 类有一个弧度和半径。它是一个类,所以对系统输入来说既可以很容易地使用弧          */
 /* 度,又可以很容易地使用半径,而且也能够改变算术的精度(float, double 或 int)。        */
 /* ------------------------------------------------------------------------ */

class Angle {
friend double cos (Angle);
friend double sin (Angle);
public:
   Angle(double d)        { radians = d; }
   Angle ()               { radians =0; }
   Angle (const Angle& a){ radians = a.radians; }
   inline Angle &operator = (const Angle& a) {
      radians = a.radians;
      return *this;
   }
   Angle operator+ (const Angle&) const;
private:
   double  radians;
};

/*------------------------------------------------------------------------ */
/* Coordinate 类描述了在笛卡儿坐标上的一个点,                                 */
/* 它可以是微米、像素、英寸和其他任何单位和尺寸。                                 */
/*------------------------------------------------------------------------ */

class Coordinate {
public:
   inline long x() const          { return x_rep; }
   inline long y() const          { return y_rep; }
   Coordinate &operator = (Coordinate &c) {
      x_rep = c.x_rep;
      y_rep = c.y_rep;
      return *this;
   }
   Coordinate(long, x, long y) {
      x_rep = x; y_rep = y;
   }
   Coordinate(const Coordinate &c) {
      x_rep = c.x_rep;
      y_rep = c.y_rep;
   }
   Coordinate()       { x_rep = y_rep = 0;}
   ~Coordinate ()     {}

   void rotateAbout (Coordinate, Angle);
 private:
```

```
    long x_rep, y_rep;
};

/*-------------------------------------------------------------------------*/
/* 这些函数重载了常规的三角函数，所有三角函数都可用 Angle 作为调用的自变量。          */
/* 因为重载不在返回值的基础上运作（只能在自变量类型基础上运作），所以三角函数 atan，   */
/*需要返回类型 Angle 的一个对象，不能被动重载，并且必须用整个新函数来建立。           */
/*-------------------------------------------------------------------------*/

inline double cos (Angle a)       { return ::cos (a.radians); }
inline double sin (Angle a)       { return ::sin (a.radians); }
inline Angle Angleatan(double t) {
    Angle a (::atan(t));return a;
}

/*-------------------------------------------------------------------------*/
/* Color 类为图形包使用而封装了颜色编码的实现：它的内部能够用低层接口子程序访问，而高层   */
/* 子程序在更通用的层次上处理。这就使得进入端口可能跨越不同机器的平台。闪光（Flashing）  */
/* 是颜色的属性。通常采用常数 Black 和 White 作为全局颜色对象的预先说明。              */
/*-------------------------------------------------------------------------*/

enum Flash { FlashBlack, FlashWhite, FlashOff };

class Color {
public:
    inline double red()       { return red_rep; }
    inline double green()     { return green_rep; }
    inline double blue()      { return blue_rep; }
         Color(double r = 0.0, double g = 0.0,
               double b = 0.0, Flash f = FlashOff) {
      red_rep = r;
      green_rep = g;
      blue_rep = b;
      flashing = f;
    }
    inline Color& operator = (const Color& C) {
       red_rep = c.red_rep;
       green_rep = c.green_rep;
       blue_rep = c.blue_rep;
      return *this;
    }
         ~Color()    { }
private:
     double red_rep, green_rep, blue_rep;
     Flash flashing;
};

const color Black, White=Color(1.0, 1.0, 1.0);

/*-------------------------------------------------------------------------*/
```

```
/* Shape 是所有图形形状的基类。它定义了在形状上所有可能的特征(操作)，                    */
/* 而且也提供了所有形状的公共操作(像移动-move)的实际实现。                                */
/*------------------------------------------------------------------------------------ */

class Shape  {
public:
   // 以 center 为中心旋转
   virtual  void  rotate(Angle a)      { }
   virtual  void  move(Coordinate xy)      {
      erase(); center_val = xy; draw();
   }
   virtual  void  draw() = 0;      // 渲染
   virtual  void  erase() {
      color = Black; draw();
   }

   virtual  void  redraw() {
      erase(); draw();
   }
    Shape();
    virtual  ~Shape();
    virtual  Coordinate origin() const { return center_val; }
    virtual  Coordinate center() const { return center_val; }
    inline   Shape   *next() const  { return next_pointer; }
    inline   Shape   *prev() const  { return prev_pointer; }
    inline   void    insert(Shape *i ) {
       i ->next_pointer = this;
       i ->prev_pointer = prev_pointer;
       prev_pointer ->next_pointer = i;
       prev_pointer = i;
    }
    inline  void  append(Shape *i)     {
                     i ->next_pointer=next();
                  i ->prev_pointer=this;
                  next_pointer ->prev_pointer=i;
                  next_pointer = i;
             }
protected:

    /*
     * 这些是替代私有部分的保护节，所以它们能用导出类的子程序访问。
    */

     Coordinate  center_val;  // 标称中心
     Color  color;        // 形状的颜色

    /*
     *这些类的静态变量用于一般图形安装的窗口包（例如 X）之下。
    */
```

```
    static   Display   display;
    static   Window    window;
    static   GContext  gc;

private:
    Shape   *next_pointer;  // 指向链表的下一个指针
    Shape   *prev_pointer;  // 指向链表的前一个指针
};
/*-------------------------------------------------------------------------*/
/* Line 是特定的形状，因为它通常是所有其他形状的绘制部件。它在形状之间只有 rotateAbout  */
/* 成员函数，作为旋转其他形状基本元素使用。rotate(旋转)操作是 rotateAbout 的退化形式。  */
/* center（中心）是作为边界点的平均值来定义的：origin 是第一个边界点。                  */
/*-------------------------------------------------------------------------*/

class Line:public Shape {
public:
   void  rotate(Angle a)  { rotateAbout(center(), a); }
   void  rotateAbout(Coordinate, Angle);
   void  draw();
   inline Line &operator = (const Line &l) {
      p=l.p; q=l.q; return *this;
   }
   Line (Coordinate, Coordinate);
   Line (Line& l) {
      p = l.p; q = l.q;
      center_val = l.center_val;
   }
   Line()        {p = Coordinate(0, 0);
            q = Coordinate(0, 0); }
   ~Line()      { erase(); }
   inline Coordinate      e() const  { return q; }
   inline Coordinate origin() const  { return p; }
private:
   Coordinate  p, q;       // 线的边界点
};

/*-------------------------------------------------------------------------*/
/* Rect 由 4 条线围成。origin 位于左上角，并且它是从构造函数的第一个参数 center      */
/* 为几何形状中心来创建的，这是我们所希望的。erase 操作从 Shape 中继承过来，          */
/* Rep 替代私有部分是在使节的，所以子类 Square 能够使用 Rep。                          */
/*-------------------------------------------------------------------------*/

class Rect :public Shape {
public:
    void    rotate(Angle);
    void    draw();
    void    move(Coordinate);
         // 左上角，x尺寸，y尺寸，
         Rect (Coordinate, long, long);
         ~Rect()          { erase(); }
```

```
   inline  Coordinate   origin() const { return l1.origin(); }
protected:
   Line   l1, l2, l3, l4;
};

/*-------------------------------------------------------------------------- */
/* Ellipse 的中心就是几何形状的中心：原点(origin)是中心。Ellipse 未实现。      */
/*-------------------------------------------------------------------------- */

class Ellipse : public Shape {
public:
   void   rotate(Angle);
   void   draw();
        // 中心，长轴，短轴
        Ellipse(Coordinate, long, long);
        ~Ellipse()     { erase(); }
protected:
   long   major, minor;
};

/*-------------------------------------------------------------------------- */
/* Triangle 由 3 条线构成。它的中心是作为所有点的平均值来计算的。              */
/*-------------------------------------------------------------------------- */

class Triangle : public Shape {
public:
   void    rotate (Angle);
   void    draw();
   void    move(Coordinate);
      Triangle(Coordinate, Coordinate, Coordinate);
      ~Triangle()   { erase(); }
private:
   Line    l1, l2, l3;
};

/*-------------------------------------------------------------------------- */
/* 一个正方形是矩形的特例。它的所有成员函数都是从 Rect 继承过                 */
/* 来的：只有构造函数是定制的，并且它不做什么只调用了 Rect 的构造函数          */
/*-------------------------------------------------------------------------- */

class Square : public Rect {
public:
      Square(Coordinate ctr, long x):
        Rect(ctr, x, x)          {}
};

/*-------------------------------------------------------------------------- */
/* 文件 Shapes.c,包含了例子程序的 Shape 代码。                                */
/*-------------------------------------------------------------------------- */
```

```
#include "shapes.h"

extern "C" {
   extern void doXInitialization(Shape&);
   extern void
     XDrawLine(Display, Window, GContext, int, int, int, int);
}

/*
* CLASS ANGLE
*/

Angle
Angle::operator+ (const Angle& angle) const {
    Angle retval = angle;
    angle.radians += radians;
    return  retval;
}

/*
* CLASS COORDINATE
*/

void
Coordinate::rotateAbout (Coordinate pivot, Angle angle)
{
   long xdistance = pivot.x() - x(), distance = pivot.y() - y();
   long xdistsquared = xdistance *xdistance,
     ydistsquared = ydistance *ydistance;
   double r = ::sqrt( xdistsquared + ydistsquared );
   Angle newangle = angle +
                Angleatan(double(ydistance)/double(xdistance));
   x_rep = pivot.x() + long(r* ::cos(newangle));
   y_rep = pivot.y() + long(r* ::sin(newangle));
}

/*
* CLASS SHAPE
*/

/* 图形包的标志(flag)。              */

static int X_initialized = 0;

Shape::Shape() {
   center_val = Coordinate(0, 0);
   next_pointer = prev_pointer = 0;
   color = White;
   if(!X_initialized) {
     doXInitialization ( *this);
```

```
      X_initialized = 1;
    }
}

/*
* CLASS LINE
*/

void
Line::rotateAbout(Coordinate c, Angle angle) {
    erase();
    p.rotateAbout(c, angle);
    q.rotateAbout(C, angle);
    draw();
    }

void
Line::draw() {
    if (p.x() - q.x() && p.y() - q.y()) {
      XDrawLine (display, window, gc, p.x (),
                                        p.y(), q.x(), q.y());
    }
}
Line::Line(Coordinate p1,Coordinate p2 ) {
    p = p1; q = p2;
    center_val = Coordinate((p, x()+q.x())/2, (p.y()+q.y())/2;
    color = Color(White);
    draw();
}

/*
* CLASS RECTANGLE
*/

void
Rect::rotate(Angle angle) {
    erase();
    l1.rotateAbout(center(), angle);
    l2.rotateAbout(center(), angle);
    l3.rotateAbout(center(), angle);
    l4.rotateAbout(center(), angle);
    draw();
}

void
Rect::draw(){
    l1.draw();
    l2.draw();
    l3.draw();
    l4.draw();
```

```
}

void
Rect::move(Coordinate c) {
   erase();
   long xmoved = c.x() - center().x();
   long y moved = c.y() - center().y();
   center_val = c;
   11 = Line(Coordinate(11.origin().x() +xmoved,
                                    11.origin().y() +ymoved),
        Coordinate(11.e().x() +xmoved, 11.e().y() +ymoved) );
   12 = Line(Coordinate(12.origin().x() +xmoved,
                                    12.origin().y() +ymoved),
        Coordinate(12.e().x() +xmoved,12.e().y() +ymoved) );
   13 = Line(Coordinate(13.origin().x() +xmoved,
                                    13.origin().y() +ymoved),
         Coordinate(13.e().x() +xmoved, 13.e().y() +ymoved));
   14 = Line(Coordinate(14.origin().x() +xmoved,
                                    14.origin().y() +ymoved)
         Coordinate(14.e().x() +xmoved, 14.e().y() +ymoved));
draw ();
}

Rect::Rect(Coordinate topLeft, long xsize, long ysize) {
   Coordinate    a(topLeft);
   Coordinate    b(a.x() +xsize, a.y());
   Coordinate    c(a.x(), a.y() +ysize);
   Coordinate    d(b.x(), c.y());
   11 = Line(a, b);
   12 = Line(b, c);
   13 = Line(c, d);
   14 = Line(d, a);
   center_val = Coordinate((11.origin().x() +12.e().x())/2,
                (14.origin().y() + 14.e().y())/2);
   draw();
}

/*
* CLASS TRIANGLE
*/

void
Triangle::rotate(Angle angle) {
   erase();
   11.rotateAbout(center(), angle);
   12.rotateAbout(center(), angle);
   13.rotateAbout(center(), angle);
   draw();
}
```

```
void
Triangle::move(Coordinate c) {
   erase();
   long xmoved = c.x() - center().x();
   long ymoved = c.y() - center().y();
   center_val = c;
   l1 = Line(Coordinate(l1.origin().x() +xmoved,
                                    l1.origin().y() +ymoved),
     Coordinate(l1.e().x() +xmoved, l1.e().y() +ymoved) );
   l2 = Line(Coordinate(l2.origin().x() +xmoved,
                                    l2.origin().y() +ymoved),
     Coordinate(l2.e().x() +xmoved, l2.e().y() +ymoved));
   l3 = Line(Coordinate(l3.origin().x() +xmoved,
                                    l3.origin().y() +ymoved),
     Coordinate(l3.e().x() +xmoved, l3.e().y() +ymoved));
   draw();
}
void
Triangle::draw()
   l1.draw(); l2.draw(); l3.draw();
}
Triangle::Triangle(Coordinate a, Coordinate b, Coordinate c) {
   l1 = Line(a, b); l2 = Line(b, c); l3 = Line(c, a);
   center_val =
            Coordinate((l1.e().x() +l2.e().x() +l3.e().x())/3,
                     (l1.2().y() +l2.e().y() +l3.e().y())/3);
   draw();
}
```

附录 C

符号的 Shapes

```
//*************************************************************************** //
//                                FILE: MPTR.H                                //
//                            C++编译数据结构的说明                            //
//*************************************************************************** //

//    这些数据结构支持 C++虚拟函数机制。vptp 是通用的函数指针；它用于寻址(但不调用)成员函数。
// vvptp 是一个指向函数返回一个指向函数的指针的指针；它用于转换的一些安装习惯用法。mptr 描述
// 了虚拟函数表的入口；它可以从 C++编译系统的对象代码中衍生出来。
typedef int ( *vptp) ();
typedef vptp( *vvptp) ();
struct mptr {short d; short i; vptp f; };
```

```
//*************************************************************************** //
//                                FILE:  K.H                                  //
//                            Thing 类和 Top 类的说明                          //
//*************************************************************************** //

# ifndef_MPTR_H
#    include "mptr.h"
# endif
# define _K_H
# include <String.h>

// 依赖于机器和编译程序
const unsigned int WRDSIZE = sizeof(void *);

inline size_t
Round(size_t s) {
   return (size_t) ((s +WRDSIZE -1)/WRDSIZE) *WRDSIZE;
}

// 通常使用字符指针类型
typedef char *Char_p;
// 哑类型，简单地用于 ShapeRep 导出类的构造函数之间
// 无二义性的地方。
```

```
enum Exemplar{};
// 依赖于机器和编译程序
class Top {
public:
    // 这个类没有数据，除非_Vtbl 由编译程序提供。从这个类中派生出来的所有类保证_Vtbl 总是任何
    // 对象中的第一个元素。如果对编译程序来说不是“真”的，那么其他机制就必须找到这样的_Vtbl，
    // 并在这里实现可能需要的改变(通过 findVtblEntry)。这也是为系统内部使用留出的一个_Vtbl 槽。
    virtual ~Top() {}
    mptr *findVtblEntry(vptp);
    void update(String, String, const char *const = "");
    static void operator delete(void *p) {
        ::operator delete(p);
    }
    // doit 是一个通用目的函数，帮助用户调节更新。

    virtual Top *doit();
protected:
    Top() {}
    static void *operator new(size_t 1) {
        return ::operator new(1);
    }
private:
    // 比较两个函数指针是否等同。
    int compareFuncs(int, vptp, vptp);
};

typedef Top *Topp;

class Thing:public Top {
    // 所有“rep”域都由 Thing 导出：它为所有信件类定义了规范格式。
public:
    virtual Thing *type() { return this; }
    Thing() {}
    virtual Thing *cutover();        // 域更新函数
    virtual ~Thing() {}          // 析构函数
    int docutover();
};

typedef Thing *Thingp;


//*************************************************************************** //
//                                FILE: K.C                                  //
//                              Thing 类的代码                               //
//*************************************************************************** //

# include "k.h"
Thing *
Thing::cutover() {
    // cutover 函数的位置标志符。这个函数由用户提供，即在旧类数据格式和新的数据格式
```

```
    //之间一个一个地调节转换。该函数在旧类的实例上被引用，并且应当返回新类的一个实例。
    return this;
}
int
Thing::docutover() {
    // 用户可以在类数据转换上选择不转换一些对象。这个函数在以一个对象一个对象为基础上
    // 告知该对象是否要转换为新的格式，如果转换，则返回 true，如果不转换则返回 false。
    // 这主要用在多重信封类的对象共享情形：该对象只需要精确地转换一次，不是每个信封
    // 都要转换，而当共享对象被转换时，docutover 可以调节(这要看它的引用计数，等等)。
    return 1;
}


//*************************************************************************** //
//                              FILE: TOP.C                                  //
//                              Top 类的代码                                 //
//*************************************************************************** //

# include "k.h"
# include <sys/types.h>

Top *
Top::doit() {
    // 未使用函数提供了现有的、用户装入的分支，以便帮助调节类的更新。
    return 0;
}

int
Top::compareFuncs(int vtblindex, vptp, vptp  fptr) {
    // 这是比较两个函数指针抽象是否等同。第一个函数指针是用它的 vtbl 索引和函数指针中
    // 刻画的；第二个函数指针正好是一个函数指针。这些参数的使用依赖于系统；
    // 这里，第一个指针的函数地址未使用。
    return vtblindex = = (int)fptr;
}

mptr *
Top::findVtblEntry(vptp f) {
    // "this"是对象的虚拟函数表的函数指针，等同于 f 参数，并且返回它的 mptr 结构的地址
    //（虚拟函数表入口的全部内容）mpp 是 vtbl 指针的地址；我们用继承结构
    // 确保 vtbl 指针将是在每个对象的起始处(所有对象都从 Top 中导出)。
mptr ** mpp = (mptr ** ) this;

// 间接引用指针地址以获取位于这个对象开始的 vtbl 地址。
mptr * vtbl = *mpp;
printf("Top::findVtblEntry(&d):vtbl = 0x% x\n", f, vtbl);

// 浏览我们自己的 vtbl;zero 入口是结束，查找所要求的函数。
for(int i = 1; vtbl{i}.f; + + i ) {
       if (compareFuncs(i, vtbl{i}.f, f)) {
           return vtbl + i;
```

```
        }
    }
    return 0;
}

// 从 C 中说明外部 "load" 函数
extern "C" vptp load(const char *);

void
Top::update(    String filename,
                String fname,
                const char *const TypedefSpec) {

    // 从文件的文件名和把此文件装入到内存中来获得函数 fname。最后的参数指定为该函数的指针
    // 类型；如果该函数是重载的，则是必需的，且相应的虚拟函数被更新。只有虚拟函数才能被更新。
    const String temp = "/tmp";

    printf("Top::update(\"%s\", \"%s\", \"%s\")\n",
        (const char *)filename, (const char *)fname,
        (const char *)TypedefSpec);
    String prepname = temp + "/" + "t.c";
    String Typedef, cast = "";
    if (strlen(TypedefSpec)) {
        Typedef = String("//TYPE used to disambiguate\
        overloaded function\n\t\t\ttypedef ") + TypedefSpec;
    } else {
        Typedef = "typedef vptp TYPE";
        cast = "(vptp)";
}
    // 构造 prepname:辅助函数返回函数被更新的地址。
    FILE *tempFile = fopen(prepname, "w");
    fprintf(tempFile, "#\
        include \"includes.h\"\n\
        extern vptp functionAddress() {\n\
            % s;\n\
        TYPE retval = %s&%s;\n\
        return (vptp)retval;\n\
        }\n",
        (const char *)Typedef,
        (const char *)cast,
        (const char *)fname);
        fclose(tempFile);
        // 编辑辅助函数
        String command = String("DIR='pwd'; cd") + temp + ";\
            CC + e0 -ISDIR -c -g" + prepname;
        system(command);
        unlink(prepname);
        String objectname =
            prepname(0, prepname.length() -2) + ".o";
        // 装入 helper 函数，检索装入返回新函数的地址。
```

```
        vvptp findfunc = (vvptp)load(objectname);
        unlink(objectname);
        printf("Top::update:calling findVtblEntry(%d)\n",
                        ( * findfunc) ());
        // 现在在这个类Vtbl类查找正确的vtbl的入口。对找到的函数findVtblEntry调用辅助函数。
        mptr * vtblEntry = findvtblEntry((*findfunc) ());
        // 现在，装入该函数的新版本，并在vtbl入口中存储它的地址。
        printf("Top::update:old vtblEntry ->f = 0x%x\n",
            vtblEntry ->f);
        printf("Top::update: calling load(\"%s\") \n",
            (const char *) filename);
        vtblEntry ->f = load(filename);
        printf("Top::update: complete, new vtblEntry ->f = 0x%x\n",
            vtblEntry ->f);
    }


//*************************************************************************//
//                          FILE: COORDINATE.H                             //
//                         结构Coordinate的接口                            //
//*************************************************************************//
# define_COORDINATE_H
// 这是一个简单的公有类，表示在图形屏幕上的笛卡儿坐标，
Struct Coordinate {
    int x, y;
    Coordinate(int xx = 0, int yy =0) {x = xx; y = yy; }
};


//*************************************************************************//
//                          FILE: SHAPE.H                                  //
//                          Shape类的接口                                  //
//*************************************************************************//

// Shape类是整个图形包的用户接口——所有其他的类就作为内部的实现。
# define _SHAPE_H
#    include "k.h"
# endif
# include "List.h"
// 转发引用说明
class ShapeRep;
class Shape:public Top {  // 在k.h中定义了Top
public:
    // 对rep转发所有的操作
    ShapeRep *operator ->() const {
        return rep;
    }
    // 构造函数和传统规范格式
    Shape();
    Shape(shapeRep&);
```

```
    ~shape();
    Shape(Shape& x);
    Shape& operator = (Shape& x);
    // 产生信件对象的类型
    Top *type();

    // 替代一个具有新版本的样本，由用户所提供的 cutover 函数更新它的实例。
    void dataUpdate(Thingp&, const Thingp);
    // 通用目的函数可以由用户装入(无论要做什么)。
    Top *doit();
    // Shape 类的垃圾收集。
    void gc();
    // 注册和未注册具有 Shape 的样本子程序。
    void Register(ShapeRep *);
    void UnRegister (ShapeRep *);
    // 初始化类变量
    static void init();  // 类初始化
private:
    friend ShapeRep;
    // 仅从 ShapeRep 中调用 operator new。
    // 未使用 operator delete。
    static void *operator new(size_t s) {
        return ::operator new(s);
    }
    static void operator delete(void *) {}
    // 保留所有实例自己的列表
    static List<Topp> *allShapes;
    // 保留从 ShapeRep 中派生的所有样本的列表。
    static List<Thingp> *allShapeExemplars;
    // 指向实际 Shape 事务结束的指针。
    ShapeRep *rep;
};
// 动态地指向 Shape 样本的指针，用 Shape::init 创建
extern Shape *shape;        // 样本
// 用户不应该必须知道，或包含 ShapeRep 的头文件。
# ifndef _SHAPEREP_H
#    include "ShapeRep.h"
# endif


//*************************************************************************//
//                              FILE: SHAPE.C                              //
//                               Shape 类的代码                            //
//*************************************************************************//

#include "Shape.h"
#ifndef _SHAPEREP_H
#   include "ShapeRep.h"
#endif
#ifndef _TRIANGLE_H
```

```
#   include "Triangle.h"
#endif
#ifndef _RECTANGLE_H
#   include "Rectangle.h"
#endif
#include "List.h"

// Shape 类为所有 Shape 做了许多内存管理工作。对所有现存的 Shape 实例进行处理。
List<Topp>  *Shape::allShapes = 0;
// 由 ShapeRep 类的子类构成的所有样本列表。
List<Thingp>  *Shape::allShapeExemplars = 0;
// Shape 样本。
extern Shape  *Shape = 0;

void
Shape::init() {
    // 为 Shape 初始化所有全局数据结构，主要是能够调节初始化的次序。
    allShapes = new List<Topp>;
    allShapeExemplars = new List<Thingp>;
    Shape = new Shape;
    // 调整 ShapeRep 类型的初始化
    ShapeRep::init();
    Triangle::init();
    Rectangle::init();
}
Top *
Shape::doit() {
    // 能够由用户重新装入并作为实用工具使用的函数。
    return 0;
}

Shape::Shape() {
    // 缺省的形状构造函数
    Topp tp = this;
    // 在 allShapes 中的列表日志
    allShapes ->put(tp);
    // 我们不知道 Shape 是什么类型；现在就设 Shape 类型为哑的。
    rep = new ShapeRep;
}

Shape:: ~Shape() {
    Listiter <Topp> tp = *allShapes;
    Topp t;
    for (tp.reset(); tp.next(t); ) {
        if (t = = (Thingp)this) {
            tp.remove_prev(t);
            break;
        }
    }
    if  (allShapes ->length() = = 0) {
```

```
        // 最后的垃圾收集。
        gc();
    }
}

Shape::Shape(Shape &x) {
    // 拷贝构造函数
    Thingp tp = (Thingp) this;
    // 带有 allShapes 列表的日志
    allShapes ->put(tp);
    // 正好指向参数的 rep
    rep = x.rep;
}

Shape::Shape(ShapeRep &x) {
    // 从 ShapeRep 中建立一个 Shape(形状)：主要是把用 Triangle, Rectangle 等
    // 建立的 ShapeRep 转换到 Shapes，再返回给用户(用户绝不会看见内部的类)
    Topp tp = this;

    // 带有 allShapes 的日志
    allShapes ->put(tp);
    // 注意：no ->ref();   调用者必须产生拷贝！
    rep = &x;
}
Shape&
Shape::operator = (Shape &x) {
    // Shape 的赋值，不用担心 rep 的指向：
    // 垃圾收集将会获得。
    rep = x.rep;
    return *this;
}

Top *
Shape::type() {
    // Shape 的类型是它自己的样本
    return shape;
}

void
Shape::dataUpdate(Thingp &oldExemplar,
                                const Thingp newExemplar) {
    // 这个函数允许类替换 ShapeRep 的子类。假定所有虚拟函数已经重新编译，并且根据新类
    // 定义来增量式地装入，也假定程序员提出了一个转换函数。当引用形状的旧的实例时，
    // 就返回指向语义上等价的拷贝的指针。
    ShapeRep *savedExemplar = (ShapeRep *) oldExemplar;
    Topp tp = 0;
    // 由迭代获取静态拷贝：我们不要去捕获新添加的事情！拷贝将从这个成员函数返回回收。
    List <Topp> staticCopy = *allShapes;
    // 改变样本
    oldExemplar = newExemplar;
```

```
    // 对这个样本转换所有的子对象
    Listiter<Topp>shapeIter = staticCopy;
    for (shapeIter.reset(); shapeIter.next(tp); ) {
        Shapepointer sp = (Shapepointer)tp;
        if (sp->rep->type() = = (Thingp)savedExemplar) {
            if (sp->rep->docutover ()) {
                ShapeRep *oldrep = sp->rep;
                printf("\tchanging shape 0x%x to new format \n", oldrep);
                sp->rep = (ShapeRep* )sp->rep->cutover();
                // 这里必须人工清除旧的 rep:
                // 垃圾收集不能用，因为这个区域已经用光了。
                oldrep->ShapeRep:: ~ ShapeRep();
            }
        }
    }

    // 从所有样本的 Shape 列表中清除这个样本，它不再作为这个抽象基类样本的一部分来参与。
    UnRegister(savedExemplar);
}

void
Shape::gc() {
    // Shape 垃圾收集器：对 ShapeRep 的所有子集调节无用对象的收集。使用 Baker 算法。
    // 首先的 Baker 一半：标记所有可达到的形状。
    Listiter<Topp> shapeIter = *allShapes;
    shapeIter.reset();
    for (Topp tp = 0; shapeIter.next(tp); ) {
        Shapepointer sp = (Shapepointer)tp;
        if (sp->rep) {
            sp->rep->mark();
        }
    }

    // 第二个 Baker 一半：扫描。使用自己的 gc 成员函数，
    // 让每个单个的子类型扫描自己的池。
    Listiter<Thingp> shapeExemplarIter = *allShapeExemplars;
    shapeExemplarIter.reset();
    for (Thingp anExemplar = 0;
                shapeExemplarIter.next(anExemplar); ) {
        ShapeRep *thisExemplar = (ShapeRep *)anExemplar;
            thisExemplar ->gc(0);
    }

    // 来回扫描交换的 Baker 空间。
    ShapeRep::FromSpace ^= 1;
    ShapeRep::ToSpace ^= 1;
}

void
Shape::Register(ShapeRep *s) {
```

```
    // 用自己的 Shape 注册的 ShapeRep 样本子程序
    Thingp tp = s;
    allShapeExemplars ->put(tp);
  }

void
Shape::UnRegister(ShapeRep *s) {
    // 允许一个本身未注册的样本
    //（当它是用一个新版本替代时）
    Thingp tp = 0;
    Listiter <Thingp> shapeIter = *allShapeExemplars;
    for ( shaPeIter.reset(); shapeIter.next(tp); ) {
        if (tp = = (Thingp)s) {
            shapeIter.remove_prev(tp);
        }
    }
}


//************************************************************************ //
//                              FILE: SHAPEREP.H                            //
//                              ShapeRep 类的接口                           //
//************************************************************************ //

#define _SHAPEREP_H
#ifndef _SHAPE_H
#   include "Shape.h"
#endif
#ifndef _COORDINATE_H
#   include "Coordinate.h"
#endif

class ShapeRep : public Thing {
public:
    /* 这里是所有用户定义的操作符。注意，因为使用了 operator ->,所以这个特性不是在 Shape 类
     * 中模拟。但是，Shape 的 rep 域必须是合适的类型。赋值操作不在这里，而在 Shape 类中。
     * return_type 既应当是基本数据类型，或是 Shape 类型，或是 Shape&类型，又可以是具体数据类型。
      */
    virtual void rotate(double);
    virtual void move(Coordinate);
    virtual void draw();
    virtual void erase();

    // 构造函数
    Shape make(Coordinate, Thingp);
    Shape make(Coordinate, Coordinate, Thingp);
    Shape make(Coordinate, Coordinate, Coordinate, Thingp);

    // 这是在导出类中的重写，其缺省动作是调用上面其中一个，Thingp 有合适的缺省动作。
    virtual Shape make();
```

```
    virtual Shape make(Coordinate);
    virtual Shape make(Coordinate, Coordinate);
    virtual Shape make(Coordinate, Coordinate, Coordinate);
    // 标记这个对象的子程序给 Baker 算法来使用。
    virtual void mark()
    // 返回一个指向对象的样本的指针类型。
   Thing *type ();

    // 执行垃圾收集。如果自变量为零，则扫描当前的池；如果自变量不为零，
    // 则建立一个新池，使之能容纳已指定好大小的对象，并忽略旧池。
    virtual void gc(size_t = 0);

    // 类构造函数
    ShapeRep();
    ~ShapeRep();
    ShapeRep(Exemplar);

    // 用于调整静态、专门类信息的初始化顺序。
    static void init();
protected:
    friend class Shape;
    Coordinate center;

    // 这是“类型域”。
    Top *exemplarPointer;

    // 内存管理状态变量：空间是“从空间”或“到空间”；gcmark 是 Baker 算法下标记对象的标记位；
    // 使用中(inUse)的意思是这个对象已经返回到由调用者使用，但也不会用垃圾收集器回收。
    unsigned char space:1, gcmark:1, inUse:1;
protected:
    // 在扫描阶段，为 Baker 垃圾收集执行公共的处理。

    static void gcCommon(size_t nbytes,
        const size_t poolInitialized,
        const int PoolSize, Char_p &h);

    // 这两个“常数”指定了 Baker 算法的的从空间和到空间。
    // 它们交换了每个垃圾收集周期。
    static unsigned char FromSpace, ToSpace;

    // 通用内存管理操作符，使用 gc 在池中操作。
    static void *operator new(size_t);
    static void operator delete(void *);

    // 使用无类型形状作为缺省返回值。
    static Shape *aShape;
};
```

```
//*************************************************************************** //
//                              FILE: ShapeRep.c                              //
//                              ShapeRep 类的代码                             //
//*************************************************************************** //

// ShapeRep 是一个用 Shape 作为信封的所有信件类的基类，
// 它是所有 Shape 智能的 ShapeRep 的导出类。

#include "ShapeRep.h"
#ifndef _RECTANGLE_H
#   include "Rectangle.h"
#endif
#ifndef_TRIANGLE_H
#   include "Triangle.h"
#endif

//“常数”作为退化的返回值使用。
Shape *ShapeRep::aShape = 0;

// 位(bit)指示了 Baker 算法的从空间和到空间，在每个周期中交换执行。
unsigned char ShapeRep::FromSpace = 0, ShapeRep::ToSpace = 1;

void ShapeRep::init() {
    // 初始化类数据结构。
    aShape = new Shape;
}

void*
ShapeRep::operator new(size_t 1) {
    // 在导出类中重写
    return ::operator new(1);
}

void
ShapeRep::operator delete (void *p)
    //在导出类中重写
    ::operator delete (p);
}

Thing *
ShapeRep::type() {
    // ShapeRep 的任何导出类的类型保留在 exemplarPointer 中。
    return (Thingp)exemplarPointer;
}

Shape
ShapeRep::make() {
    // 缺省的 ShapeRep 制造者
    return *aShape;
}
```

```
Shape
ShapeRep::make(Coordinate c1, Thingp) {
    // 一个约束：Point 对象
    return point ->make(c1);
}

Shape
ShapeRep::make(Coordinate c1, Coordinate c2, Thingp type) {
    // 有两个约束的 Shape 对象的样本构造函数：Lines 和 Rectangle.
    return((ShapeRep *)type ) ->make (c1, c2);
}

Shape
ShapeRep::make(Coordinate c1, Coordinate c2, Coordinate c3,
            Thingp type) {
    // 有 3 个约束的 Shape 对象的样本构造函数：Triangle, Arc 和 Parallelogram。
    return ((ShapeRep *)type) ->make(c1, c2, c3);
}

Shape
ShapeRep::make(Coordinate c1) {
    // 由一个坐标缺省所构成的 Shape 返回是一个 Point。
    return make(c1, point);
}

Shape
ShapeRep::make(Coordinate c1, Coordinate c2) {
    // 由两个坐标缺省所构成的 Shape 返回的是 Rectangle。
    return make(c1, c2, rectangle);
}

Shape
ShapeRep::make(Coordinate c1, Coordinate c2, Coordinate c3) {
    // 由 3 个坐标缺省所构成的 Shape 返回的是 Triangle。
    return make(c1, c2, c3, triangle);
}

void
ShapeRep::gcCommon(size_t nbytes, const size_t poolInitialized,
          const int PoolSize, Char_p &heap) {
    // 垃圾收集对所有 ShapeRep 是通用的，这是 Baker 算法扫描的第二个一半，
    // 它用导出类 gc 子程序调用，该子程序由用户直接用合适的参数调用。

    // 计算 s 是作为该对象的大小。如果指定了 nbytes,并使用之；即指定了在尺寸上改变
    // (或在初始化时获取)。此外，使用的这个值存入 podInitialized。
    size_t s = nbytes?nbytes:poolInitialized;
    // 只入不舍地调整约束。
    size_t Sizeof = Round(s);
```

```
// 如果给定非零参数，则它就是对象的尺寸；即表示抛弃旧
// 池而分配新池。这是在启动之时和类更新之后需要做的事情。
if (nbytes) heap = new char [PoolSize *Sizeof];

ShapeRep *tp = (ShapeRep *)heap;

// 扫描该池
for (int i = 0; i < PoolSize; i + +) {
    switch (nbytes) {
    case 0:    // 正规的垃圾收集情况

      if (tp ->inUse) {
          if (tp ->gcmark || tp ->space != FromSpace) {
             // 不再扫描
             tp ->space = ToSpace;
          } else if (tp ! = tp ->type()) {
                //对象需要回收
                tp ->ShapeRep:: ~ShapeRep();
                tp ->inUse = 0;
                printf("ShapeRep::gcCommon");
                printf("Reclaimed Shape object %c \n",
                'A' + (((char *)tp - (char *)heap)/Sizeof));
             }
           }
           break;
       default:  // 初始化内存场所
           tp ->inUse = 0;
           break;
       }
       tp ->gcmark = 0;
       tp = (ShapeRep *) (Char_p(tp) + Sizeof);
    }
}

ShapeRep::ShapeRep(Exemplar) {
    // 建立 ShapeRep 的样本的构造函数(不是很重要)。
    // 撤销样本是用无二义性的 this 和缺省构造函数的哑类型。
    exemPlarPointer = this;
    shape ->Register(this);
}

void
ShapeRep::rotate(double) {
    // 动作像纯虚拟函数
    printf ("ShapeRep::rotate(double)\n");
}

void
ShapeRep::move(Coordinate) {
    // 动作像纯虚拟函数
```

```
    printf ("ShapeRep::move(Coordinate) \n");
}

void
ShapeRep::draw() {
    // 现在，动作像纯虚拟函数。绘制的子程序将打印类型的名字。
    printf("<Shape>");
}

void
ShapeRep::erase() {
    // 动作像纯虚拟函数
    printf("ShapeRep::erase()");
}

void
ShapeRep::gc(size_t) {
    // 动作像虚拟函数
}

void
ShapeRep::mark() {
    // 在 Baker 标记阶段使用 Shape
    gcmark = 1;
}

ShapeRep::ShapeRep() {
    // 缺省构造函数
    exemplarPointer = this;
    gcmark = 0;
    space = ToSpace;
    inUse = 1;
}

ShapeRep::~ShapeRep() {
    // 析构函数什么也没有做。
}


//*************************************************************************//
//                              FILE: LOAD.C                               //
//                          C 函数的装入函数的代码                          //
//*************************************************************************//

# include < a.out.h >
# include <fcntl.h>
# include "mptr.h"
# include "String.h"

static String symtab;
```

```
static char y = 'a';

extern "C" vptp load(const char *filename) {
    // 把第一个连接编辑指定文件装入到新的 a.out，使用以前的 a.out 文件作为还原符号引用的基。
    // 然后装入打开的新文件，指定它的.text 和.data 的大小，并且把它们读到最新的内存分配块中。
    // 其函数作为动态装入器，指定外部增量式连接编辑器的工作。C 的连接工作就好像是从 C 调用一样。

    int errcode = 0;
    String newfile;
    char buf[256];
    long adx, oadx;
    unsigned char *ldadx;
    struct exec Exec;
    int fd, wc;

// 首先要使用合理的缺省；a.out 是文件提供的符号表。每次我们
// 增量式地装入一个，改变生成连接编辑文件名，再清除旧文件。
if (!symtab.length()) {
    symtab = "a.out"; newfile = "b.out";
} else {
    symtab = String(+ +y) + ".out";
    newfile = String(y +1) + ".out";
}

// 寻找当前内存的高位和填充，所以内存高位是页的边界。
     oadx = (long)sbrk(0);
     adx = oadx + PAGSIZ - (oadx% PAGSIZ);

// 建立装入命令，根据当前的 a.out 执行提供的.o 的增量
// 式连接编辑，指定在内存高位上的新的连接代码。
sprintf(buf, "ld -N -Ttext %X -A%s %s -O %s",
    adx, (const char *)symtab, filename,
    (const char *)newfile);
printf ("<%s> \n", buf);
if ((errcode = system(buf))! = 0) {
    printf("load: link edit returned error code %d \n",
        errcode);
}
if (symtab ! = "a.out") unlink(symtab);
// 打开并装入内存之中
fd = open(newfile, O_RDONLY);
if (fd < 0) {
     printf("load:open of \"%s\"failed\n", newfile);
     return;
}

// 读重定位的文件头，以获得文本和数据的大小。
read(fd, (char *)&Exec, sizeof(struct exec));

// 现在执行内存填充（pad），并且分配新的程序文本的空间。
```

```
sbrk(int(PAGSIZ - (oadx%PAGSIZ)));
ldadx = (unsigned char *)
    sbrk(int(Exec.a_text + Exec.a_data + Exec.a_bss));
// 把最新的连接文件读入地址计算上的运行处理中。
wc = read(fd, (char *)ldadx,
    int (Exec.a_text + Exec.a_data));
close(fd);

// 返回装入地址
return (vptp) ldadx;
}


//*************************************************************************//
//                                FILE : MAIN.C                            //
//                            几何形状例子的典型驱动代码                   //
//*************************************************************************//

# include <a.out.h>
# include <fcntl.h>
# include "Shape.h"
# ifndef _COORDINATE_H
#    include "Coordinate.h"
# endif
extern void doClassUpdate();
extern int compile(const String & fileName);
extern int mkfile(const String & fileName,
                          const String & contents);

int main2() {
    Shape::init();
    Coordinate p1, p2, p3;
    Shape object = ( *shape) ->make(p1, p2, p3);
    printf("object is"); object ->draw(); printf("\n");

    // 演示虚拟函数更新
    object ->move(p1);
    compile("v2Triangle.c");
    String include = "includes.h";
    mkfile(include,
        "#include \"k.h\"\n#include \"v2Triangle.h\"\n");
    object ->update ("v2Triangle.o", "Triangle::move");
    object ->move(p1);
    doClassUpdate();
    object ->move(p1);
    {
        Shape object3 = ( *shape) ->make(p1, p2, p3);
        printf("object3 is"); object3 ->draw(); printf("\n");
    }
 printf("main:making object2 \n");
```

```
    Shape object2 = ( *shape) ->make (p1, p2, p3);
    shape ->gc();      // 现在执行 gc
    printf("object2 is ");object2 ->draw(); printf("\n");
printf("main: made object2, calling object2 ->move \n");
    object2 ->move(p1);
    shape ->gc();      // 现在执行 gc
printf("exiting\n");
    return 0;
int main() {
    int retval = main2();
    shape ->gc();
    return retval;
}

void
doClassUpdate() {
    extern Shape *triangle;
    const String include = "includes.h";
    mkfile(include,
        "#include \"k.h\"\n#include \"v2Triangle.h\"\n");

    compile("v3TriangleA.c");
    ( *triangle).update("v3TriangleA.o", "Triangle::make"
        "Shape (Triangle:: *TYPE) ()");

    compile("v3TriangleB.c");
    ( *triangle).update("v3TriangleB.o", "Triangle::make",
        "Shape (Triangle:: *TYPE) \
            (Coordinate, Coordinate, Coordinate)");

    compile("v3TriangleMove.c");
    ( *triangle).update("v3TriangleMove.o", "Triangle::move");

    compile("v3TriangleCutover.c");
    ( *triangle).update("v3TriangleCutover.o", "Triangle::cutover");

    mkfile(include, "#include \"k.h\"\n\
        #include \"v3 Triangle.h \"\n");

    mkfile ("v3doit.c", "#include \"k.h\"\n\
        #include \"v3 Triangle.h\"\n\
        Top *Shape::doit() {\n\
             printf(\"v3 Shape::doit (new) called \\n\"); \n\
             Thingp Ttriangle = triangle; \n\
             shape ->dataUpdate(Ttriangle, \n\
                 new Triangle(Exemplar(0))); \n\
             triangle = (ShapeRep *) Ttriangle; \n\
             printf(\"Shape::doit:did data update \\n\"); \\
             return 0; \n\
        }\n\n\
```

```
        Triangle::Triangle(Exemplar e):ShapeRep(e) {}\n");

    compile("v3doit.c");
    printf("doClassUpdate:\
        calling shape ->update (\"v3doit.o\",\
        \"Shape::doit \") \n");
    shape ->update("v3doit.o", "Shape::doit");
    shape ->doit();
    shape ->gc();      //现在执行gc
    unlink("v3doit.c");
    unlink("v3doit.o");
}

#include <sys/stat.h>
int compile(const String& fileName) {
    struct stat dotC, dotO;
    string fileNameDotO =
           fileName(0, fileName.length() -2) + ".o";
    stat(fileName, &dotC);
    stat(fileNameDotO, &dotO);
    if(dotC.st_mtime <dotO.st_mtime) {
       printf ("\"% s\"is up to date \n", (const char *)fileName);
       return 0;
    } else {
        String command = String ("CC +e0 -c -g") + fileName;
        printf("compile: <%s> \n", (const char * )command);
        return system(command);
    }
}

extern int mkfile(const String & fileName,
                                const String & contents) {
    FILE * inc = fopen(fileName, "w");
    printf("mkfile:creating < %s > \n", (const char * )fileName);
    fprintf(inc, (const char * )contents);
    return fclose (inc);
}


//************************************************************************ //
//                              FILE:TRIANGLE.H                            //
//                              Triangle类的接口                           //
//************************************************************************ //

// 这是对几何形状、三角形的代码实现语义的接口。
# define _TRIANGLE_H
# ifndef _SHAPEREP_H
#    include "ShapeRep.h"
# endif
# ifndef _COORDINATE_H
```

```
#    include "Coordinate.h"
# endif

class Triangle: public ShapeRep {
public:
   // 样本构造函数
   Shape make();
   Shape make(Coordinate, Coordinate, Coordinate);
   // 内存管理
   void  *operator new (size_t);
   void operator delete(void *);
   void gc(size_t = 0);
   // 用户应用的语义
   void draw();
   void rotate(double);
   void move(Coordinate);
   // 类子程序
   Triangle(Exemplar);
   Triangle();
   static void init();
private:
   // 这些从未调用
   Shape make(Coordinate);
   Shape make(Coordinate, Coordinate);
private:
   // 实例状态变量
   Coordinate p1, p2, p3;
private:
   // 内存管理数据
   static char *heap;
   static size_t poolInitialized;
      enum { PoolSize = 10 };
   }
   // Triangle 样本指针说明
   extern ShapeRep *triangle;


//************************************************************************ //
//                              FILE: TRIANGLE.C                            //
//                               Triangle 类的接口                          //
//************************************************************************ //

#include "Triangle.h"
// 指定类的变量
ShapeRep *triangle = 0;
char *Triangle::heap = 0;
size_t Triangle::poolInitialized = 0;

void
Triangle::init() {
```

```
    // 初始化指定三角形状态和全局变量，
    // 某些数据结构初始化首先要调用 ShapeRep::gcCommon
    triangle = new Triangle(Exemplar(0));
}

Shape
Triangle::make()
{
    // 构造缺省(退化)的三角形
    Triangle *retval = new Triangle;
    retval ->p1 = retval -> p2 = retval -> p3 = Coordinate(0, 0);
    retval ->exemplarPointer = this;
    return *retval;
}

Shape
Triangle::make(Coordinate pp1, Coordinate pp2, Coordinate pp3)
{
    // 建立并返回一个新的三角形
    Triangle *retval = new Triangle;
    retval ->p1 = pp1;
    retval ->p2 = pp2;
    retval ->p3 = pp3;
    retval ->exemplarPointer = this;
    return *retval;
}

void
Triangle::gc(size_t nbytes) {
    // 把三角形内存数据结构传送到基类中的公共的垃圾收集(扫描)子程序中
    gcCommon(nbytes, poolInitialized, PoolSize, heap);
}

void
Triangle::draw(){
    // 至此，打印在该池中相应的三角形位置的名字，
    int Sizeof = poolInitialized? poolInitialized:
        Round(Sizeof(Triangle));
    printf("<Triangle object %C>",
        'A' + (((char *)this - (char *)heap)/Sizeof));
}

void
Triangle::move(Coordinate) {
    //...
}

void
Triangle::rotate(double) {
    //...
```

```
}

void*
Triangle::operator new(size_t nbytes) {
    // 如果池未被初始化，或者如果我们正在更新 Triangle 类，
    // 那么就给出垃圾收集器的控制。
    if (poolInitialized - nbytes){
        gcCommon(nbytes, poolInitialized, PoolSize, heap);
        poolInitialized = nbytes;
    }
    // 寻找一个空闲
    Triangle  *tp = (Triangle *)heap;
    // 需要添加内存耗尽的测试
    while (tp ->inUse) {
        tp = (Triangle *) (((char *)tp) + Round(nbytes));
    }
    // 初始化合适的内存位
    tp ->gcmark = 0;
    tp ->inUse = 1;
    return(void *)tp;
}

void Triangle::operator delete(void *) {
    // 这个应该从未调用过，但如果是新的，则 C++就认为它在这里。
}

Triangle::Triangle(){}   // 这里是 Size 和 Vptr 的信息
Triangle::Triangle(Const Triangle& t) {
    // 拷贝构造函数。
    p1 = t.p1;
    p2 = t.p2;
    p3 = t.p3;
}

Triangle::Triangle(Exemplar e) : ShapeRep(e) {
    // 建立一个三角形样本。
}

Shape
Triangle::make(Coordinate) {
    // 哑函数——应该从未调用过。
    return *aShape;
}

Shape
Triangle::make(Coordinate, Coordinate) {
    // 哑函数——应该从未调用过。
    return *aShape;
}
```

```
//*************************************************************************** //
//                              FILE:  V2TRIANGLE.C                          //
//                         Triangle 类的修改代码(版本 2)                      //
//*************************************************************************** //

#include "v2Triangle.h"

void
Triangle::move(Coordinate)
{
   printf("version 2 Triangle::move of size % d\n",
       sizeof( *this));
}


//*************************************************************************** //
//                              FILE: V2TRIANGLE.H                           //
//                         Triangle 类的修改接口(版本 2)                      //
//*************************************************************************** //

# define _TRIANGLE_H
# ifndef _SHAPEREP_H
#   include"ShapeRep.h"
# endif
# ifndef _COORDINATE_H
#   include"Coordinate.h"
# endif
class Triangle:public ShapeRep {
public:
   //就好像是 Triangle.h, 除非新的 Triangle::move 的定义被建立。
   Shape make();
   Shape make(Coordinate, Coordinate, Coordinate);
   Triangle();
   void draw();
   void move(Coordinate);
   void rotate(double);
   void *operator new(size_t);
   void operator delete(void *);
   void gc(size_t = 0);
   Triangle(Exemplar);
   Static void init();
private:
   static void poolInit(size_t);
   Shape make(Coordinate) {return *aShape; }
   Shape make(Coordinate, Coordinate) { return *aShape; }
   Coordinate p1, p2, p3;
private:
   static char *heap;
   static size_t poolInitialized;
```

```
    enum { PoolSize = 10 };
};

extern ShapeRep *triangle;


//*************************************************************************** //
//                              FILE: V3TRIANGLE.H                            //
//                          Triangle 类的版本 3 的数据结构                    //
//*************************************************************************** //

# define _TRIANGLE_H
# ifndef _SHAPEREP_H
#    include"ShapeRep.h"
# endif
# ifndef _COORDINATE_H
#    include"Coordinate.h"
# endif

// 这是一个新(版本 3)三角形类的说明——它有一个颜色属性

class Triangle:public ShapeRep {
public:
    Shape make();
    Shape make(Coordinate, Coordinate, Coordinate);
    Triangle();
    void draw();
    void move(Coordinate);
    void rotate(double);
    void *operator new(size_t);
    void operator delete(void *);
    void gc(size_t = 0);
    Thing *cutover();
    Triangle(Exemplar);
    static void init();
private:
    static void poolInit(size_t);
    Shape make(Coordinate) { return *aShape; }
    Shape make(Coordinate, Coordinate) { return *aShape; }
    Coordinate p1, p2, p3;
    enum Color { Black, White } color;
private:
    static char *heap;
    static size_t poolInitialized;
    enum {PoolSize = 10};
};

extern ShapeRep *triangle;
```

```
//*************************************************************************** //
//                          FILE: V3TRIANGLEA.C                              //
//                       Triangle 类的版本 3 的代码                          //
//*************************************************************************** //

# include"v3Triangle.h"

Shape
Triangle::make()
{
   printf("Triangle::make() entered\n");
   Triangle *retval = new Triangle;;
   retval ->p1 = retval ->p2 = retval ->p3 = Coordinate(0, 0);
   retval ->exemplarPointer = this;
   color = Black;
   return Shape ( *retval);
}

// 这将使用 make()获取静态扩张。
Triangle::Triangle() {}  // 这里,size 和 vptr 是信息。


//*************************************************************************** //
//                          FILE: V3TRIANGLEB.C                              //
//                       Triangle 类的版本 3 的代码                          //
//*************************************************************************** //

# include "v3Triangle.h"

Shape
Triangle::make(Coordinate pp1, Coordinate pp2, Coordinate pp3)
{
   Triangle petval = new Triangle;
   retval ->p1 = pp1;
   retval ->p2 = pp2;
   retval ->p3 = pp3;
   retval ->exemplarPointer = this;
   return *retval;
}

Triangle::Triangle() {}  // 这里是 size 和 vptr 的信息。


//*************************************************************************** //
//                       FILE: V3TRIANGLECUTOVER.C                           //
//                  版本 3 的 Triangle 代码的转换控制代码                     //
//*************************************************************************** //

# include"v3Triangle.h"
# include"Map.h"
```

```
// ---------------------------------------------------------------------------

class v2Triangle: public ShapeRep {
public:
    Shape make();
    Shape make(Coordinate, Coordinate, Coordinate);
    v2Triangle();
    void move(Coordinate);
    void *operator new(size_t);
    void operator delete(void *);
    void gc(size_t = 0);
    void draw();
    v2Triangle(Exemplar);
    static void init();
private:
    friend Thing *Triangle::cutover(); // put in for conversion
    Static void poolInit(size_t);
    Shape make(Coordinate) { return *aShape; }
    Shape make(Coordinate, Coordinate) { return *aShape; }
    Coordinate p1, p2, p3;
    };

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

// 使用这个映像可以跟踪所有我们要求转换的旧对象，并且转换一个
// 新对象。即，如果我们要转换相同对象几次，那么我们的映像就要求
//在同一返回值上。

Map <Thingp, Thingp> objectMap;

Thing *
Triangle::cutover() {
    // 我们正返回指向被转换成三角形的指针。
    Triangle *retval = this;
    // 在旧的三角形中实例被传送
    // 旧的三角形说明被保存在名字 v2Triangle 下，
    // Triangle 类是版本 3 中的一个。
v2Triangle *old = (v2Triangle *)this;
Thingp oldtp = this;
ShapeRep *oldsr = (ShapeRep *)this;

if (objectMap.element(oldtp)) {
    // 如果我们已经转换了，就不要再转换了——只需返回
    // 旧的转换值。
    retval = (Triangle *) (objectMap [oldtp]);
} else {
    // 建立一个新的(版本 3)三角形，返回存储在几个不
    // 同种类中的指针之中。
    retval = new Triangle;
```

```
    ShapeRep *newsr = retval;
    Thingp newtp = retval;

    // 拷贝部分基类(ShapeRep)
    *newsr = *oldsr;

    // 现在建立新的对象域
    retval ->exemplarPointer = triangle;
    retval ->p1 = old ->p1;
    retval ->p2 = old ->p2;
    retval ->p3 = old ->p3;
    retval ->color = Black;

    // 存储最后转换的一个。
    objectMap[oldtp] = newtp;
  }
    return retval;
}

Triangle::Triangle() { }


//************************************************************************ //
//                          FILE: V3TRIANGLEMOVE.C                        //
//                      版本 3 的 Triangle 的移动实现的代码                //
//************************************************************************ //

# include"v3Triangle.h"

void
Triangle::move(Coordinate)
{
   printf("version 3 Triangle::move of size %d\n",
       sizeof( *this));
}
```

附录 D

C++中块结构程序设计

自顶向下设计是一种成熟的软件工程方法。在过去的 30 年中，这种设计已成为绝大多数结构设计技术的基础。面向对象设计的出现取代了这些技术，特别是在处理复杂的大型系统时。但是，诸如功能分解这样的自顶向下的设计技术仍能在容易理解的算法中有用；或者当该问题是自主时和它的解结构是可预见时，自顶向下设计技术还是相当有用的。

块结构程序设计是支持自顶向下设计的程序设计语言。附录 D 将展示如何把一些 C++习惯用法提供给块结构程序设计的有用的变体。习惯用法要求显式作用域限定，它比在 Pascal 或在 Modula-3 中显得更有必要，这应归功于 C++语言的作用域特性。

C 预处理宏是作为 C++语言的一部分来定义的。附录 D 也准备用混合 C++构造宏的方法给读者增添点情趣，以完成单独的“基本 C++语言”不可能完成的功能性的和表达性的东西。

D.1 什么是块结构程序设计

自顶向下设计似乎已经占据了软件结构技术的支配地位。通常使用的自顶向下设计是功能分解，或者是逐步提炼。早在 1970 年，Niklaus Wirth 就主张使用功能分解。在功能分解中，系统初始用高层功能特性实施，然后把此功能划分为若干个片，每片再进行同样的分解如此继续下去，最后的结果是具有能够直接实现正确语义模块的集合。

在功能分解中，数据扮演着次要角色。在每个精细的改进层次上，数据设计是对每个过程实体进行的。数据设计派生出被分解的模块设计，过程认为存入它“自己的”这些数据，数据块的生命周期是过程自己的数据的激活记录。数据块的作用域同样由过程结构来定义。

自顶向下设计由程序设计语言支持了多年。支持自顶向下设计的一个重要的语言特征是块结构程序设计。块结构程序设计语言具有作用域构造，允许过程相互嵌套。如 C 和 C++语言就称为块结构，但仅是弱的含义。虽然把局部变量从全局变量中分开，并允许在过程内嵌套说明块，但嵌套的块结构并不扩充到过程。就此种定义而言，C 和 C++都不是合格的块结构程序设计语言。反之，Algol 68、PL/I、Modula-3、Ada 和 Pascal 才是合格的块结构程序设计语言。

C++的简明、灵活的机制以及创造性的应用，提供了构造块结构程序的支柱，我们采用块结构习惯用法来调用这种风格和技术。此习惯用法引入一种限定形式，即允许邻近的作用域之间进行符号存取，然后再扩充到更一般的形式。

当你使用这种习惯用法时：块结构是非常适合一类直截了当的问题，在此问题中，算法和过程支配着数据之间的联系和结构。这种方法的问题控制流是顺序的，其问题解要求至多不超过 1 000 行代码，并且解的算法是容易理解的。

D.2 构造基本块结构的 C++程序设计

构造 C++块结构，就必须建立一个新的作用域或建立在每个函数内的名字空间，在每个函数内部能够包含它自己的函数。struct 非常适合块结构的描述。C++与 C 不同，C++的 struct 说明可以包含函数，函数是局部于该结构的。struct 能够用在对每个功能分解的封装数据设计上。我们将看到，封装使 C++块结构程序较 Algol 的程序更容易分析。

虽然 struct 提供支持块结构的语义层，但其语法较为混乱，这是与 Algol 和 Pascal 过程块结构相比较而言的。预处理宏使程序员设计更加清晰，这些宏用来描绘每个函数体中创建新的作用域的内容。我们在头文件 block.h 中置入这些宏，如下面的代码所示：

```
// 文件 block.h
# include <generic.h>
# define LocalScope(function)
   struct name2(function, ForLocalScope) { public
# define EndLocalScope } local
```

LocalScope 说明正好是建立 struct 说明的打开段。定义在 generic.h 中的 name2 宏将它的两个自变量耦合到一起，形成单一的新的名字。按照惯例，LocalScope 宏的参数应当嵌入到函数的名字之中。EndLocalScope 宏紧挨着 struct 说明，并说明一个命名为 local 的 struct 实例，用这种 struct 就可以建立上下文处理。

例 D.1 是块结构冒泡排序实现的一个例子。在模块 CompareExchange 中分解冒泡排序算法，这个模块已经在它自己的权限之内析出一个函数，被抽象在 BubbleSort 之内。BubbleSort 与 CompareExchange 的引用和谐地结合，填充了排序算法。

例 D.1：块结构冒泡排序的实现。

```
# include <block.h>
# include <iostream.h>
# include <string.h>

void BubbleSort(int n, char *records[ ],char *keys [ ])
{

   LocalScope(BubbleSort):
      int bound, t;
      void CompareExchange(int j, char *records[ ],
                                        char *keys[ ]) {
          if(::strcmp(keys[j], keys[j +1]) >0) {
             char *temp = records[j];
             records[j] = records [j+1];
             records[j + 1] = temp;
             temp = keys[j];
             keys[j] = keys[j+1];
             keys[j+1] = temp;
             t = j;
          }
      }
```

```
    EndLocalScope;

    local.bound = n;
    do{
        local.t = -1;
        for (int j = 0; j< local.bound-1; j + +) {
            local.CompareExchange(j, records, keys);
        }
        local.bound = local.t + 1;
    } while (local.t! = -1);
}
```

注意，虽然 BubbleScope 是作为一般函数来存取，但是在 BubbleSort 外部 CompareExchange 不可见。块结构提供了在过程空间中的信息隐藏，而算法细节被封装在高层抽象中。

下面是使用冒泡排序实现的例子：

```
Char *records[ ] = {
    "Stroustrup, Bjarne",
    "Lippman, Stan",
    "Hansen, Tony",
    "Koening, Andy"
  };
Char *keys[ ] = {
    "bS",
    "Stan",
    "hansen",
    "ark"
  };
int main( ) {
    for (iht i = 0; i<4; i + +) {
        Cout<<records[i]<<endl;
  }
    BubbleSort(4, records, keys);
    for(i = 0; i<4; i + +) {
        Cout <<records[i]<<endl;
  }
  return 0;
}
```

当设计一个过程式分解层次结构时，一定要在自己的头脑中尽量保留少量的管理规则。一种凭经验的做法是在 LocalScope 块内说明函数的一些局部变量。在这些块中置入“局部”变量，于是在函数建立块和功能分解的下一层的函数都产生了作用域。例如，在上述程序中，*t* 和 bound 是在 BubbleSort 的作用域中，它们用 BubbleSort 作为自己的作用域，但可以被 CompareExchange 访问。注意：在 BubbleSort 内的存取用 local 前缀和点操作符来限定。

块结构的合理性在于允许函数在作用域内访问和改变其变量值，包括函数内的局部变量。就这种意义来说，它违背了封装函数的概念。功能分解自然遍及层次结构从顶到叶的各个方向，并且低层抽象依赖于高层抽象。不过，我们不要在进行高层抽象时过分地注意低层。过程变量应当是它自己的，并且用低层来访问这些过程变量，更精细的子程序应该更容易从代码检查中发现。对深度嵌套作用域的局部变量访问似乎违背了程序演变和维护上的“最少意外的原则”。

在这里，该方法引入了对块中变量访问的限制，其用意是只在“自己的”作用域内能够可见，或者至多在一层深度内能够可见。下一层（即 CompareExchange）从当前层（即 BubbleSort）中是直接可见的，这意味着对研究作用域变量领域，外延搜索不是必要的。

注意，某些琐细的变量（如循环变量）不接受这种处理。它们不是数据设计的一部分，这些变量可以用 C++语言进行常规的处理。

另一条经验规则（作为推荐的方法）是使用参数传递，使之能从一层到另一层访问数据。即可以在嵌套块之间通过全局数据来传递数据，或者在嵌套块之间通过 LocalScope 内数据来传递数据。但是当查找这样的数据所引用的所有位置时，会引起维护上的麻烦。定位嵌套块中的数据需要帮助，但块中的数据项仍可以访问块内的任一函数。原始接口可以通过参数传递数据，并用返回值返回结果。事实上，对不是 LocalScope 组成部分的变量是必需的。例如，BubbleSort 的参数是间接地对 CompareExchange 进行访问的。为了使之能够访问，就必须要使变量的参数形式传递，就像我们看到的冒泡排序例子那样。这是一个好的设计的例子，具有好的维护定义，在模块之间有显式接口。

D.3 有深度嵌套作用域块的选择

块结构习惯用法只允许对邻近作用域访问，尽管存在一些语法上的棘手问题。大多数块结构语言都假定在它自己作用域内的符号是可见的，但在这里，块结构习惯用法不作这样的假定。这里，符号只能访问自己的作用域(如使用 local. 前缀，Bubble 能够访问 bound 和 t)，并且能从包含在此作用域（CompareExchange 中能够访问相同的符号）的下一个低层函数中访问。在 C++作用域构造的限制中，要完成块结构语言中全部嵌套作用域的概括是很困难的。但是，如果我们能够访问全部嵌套的作用域，那么就可能帮助读者找到引用符号的说明，并在 C++限定之内来实现，然后在显式作用域分解和在嵌套作用域结构之间完成习惯用法上的权衡。动态块结构的习惯用法就是这样的一种方法。

附录 D 后面附有设计简单的视频游戏子程序的部分实现，游戏的规则如图 D.1 所示，这是比冒泡排序更加典型的自顶向下设计。它的结构方块图如图 D.2 所示，在这个程序中，我们增加了一些机制，使用户能够显式地访问外部作用域。

```
SSSS
```

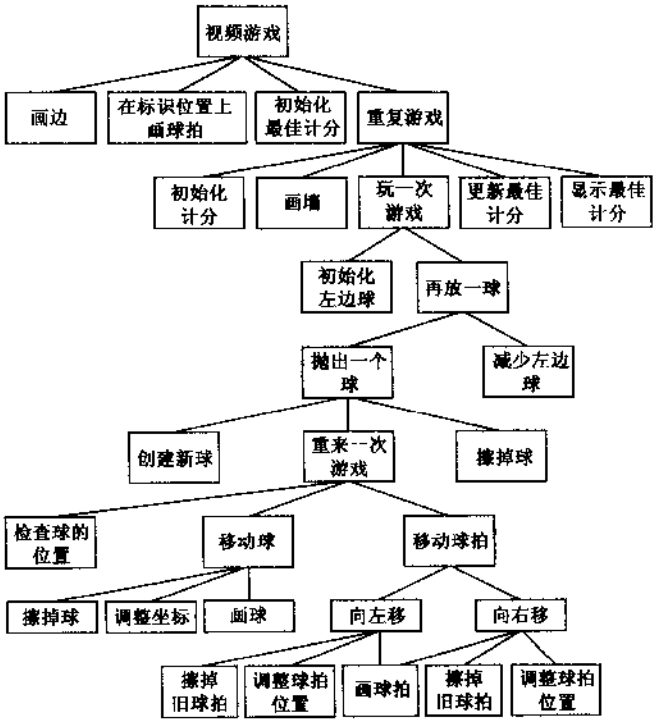


图 D.2 游戏方块图的层次结构

开始时，我们把构造未加工作用域机制置入到宏中。该作用域机制维护了动态链，或调用这个嵌套作用域结构中函数的历史表。程序员能够对任何嵌套函数作用域的链来回游动，并存取它们的任何数据或调用任何函数。我们采用让 LocalScope 宏定义作用域构造函数的技巧，即不管何时打开作用域，都能自动地调用。按照规定，构造函数的名字与 struct 的名字相同，并且通常情况下它初始化新变量，创建给定的结构类型。这个构造函数采用单个参数 this，使编译指向当前对象，所以新的 block.h 文件开始具有如下形式：

```
#include <generic.h>
#define LocalScope(function)
    struct name2 (function, ForLocalScope) {
        void *parent;
        name2 (function, ForLocalScope) (void *p)
            {parent = p;}
    public
#define EndLocalScope
    } local(this)
```

LocalScope 宏已经扩充到包含 parent 说明，parent 是一个指向父类作用域的指针，并初始化 parent 值的一个构造函数。宏引用

```
LocalScope (SomeFunction)
```

扩充的代码等价于：

```
struct SomeFunctionForLocalScope {
   void *parent;
   SomeFunctionForLocalScope(void *p){parent = p; }
}
```

EndLocalScope 宏结束了包含作用域信息的 struct 和定义一个命名为 local 的变量。变量 local 用 this 初始化（采用 EndLocalScope 的部分说明）。this 变量涉及到封装的作用域，因为内部类在封装的作用域的成员函数内。local 值通过封装作用域构造函数的一个参数来传递，这个参数建立了从给定作用域到它的封装作用域的链（即封装的 local 对象）。

最外层接受特殊的处理，因为它没有封装 LocalScope 对象。最外层 LocalScope 块仍有一层作用域在此之外——全局作用域，但没有正式的链接维护这两层。全局变量不遵循正规块作用域的限制：这些全局变量总是可以访问的，不加限制。因为最外层的 LocalScope 块不能从封装块的 this 指针中创建作用域链，它需要自己的“结束作用域”宏，对空指针设置父辈作用域指针，它是调用 EndOuterScope：

```
#define EndOuterScope local(0)
```

为了访问父辈作用域，我们提供了 parent 宏；只用一点点语法来应用 void *parent 作用域指针，并间接引用其中一个域：

```
#define Parent (type, c)
   ((name2(type, ForLocalScope) *) (c) ->parent)
```

要访问封装的作用域，必须引用有封装作用域函数名的 parent 宏，作为它的第一个参数，而用 this 作为第二个参数，其返回类型是指向作用域对象的一个指针：

```
outerProcLocalScope *scopePtr = Parent(outerProc, this);
```

parent 宏返回它的封装作用域的 this 指针。该宏能够重复应用，以成功地访问多个全局作用域：

```
PlayGameLocalScope *p = Parent(PlayGame, this);
BallGameLocalScope *p2 = Parent(BallGame, p);
int &y p2 ->ball.yposition;
```

为了了解这个作用域的工作状况，例 D.2 介绍了这个程序。

例 D.2：交叉访问多层作用域。

```
int main( ) {
   LocalScope(main):
      int a;
      int b;
      void foo( ) {
         LocalScope(foo):
            int a, b;
            void bar( ) {
               local.a = 3; local.b = 4;
               a = 5; b = 6;
               printf ("bar:a=%d, %d, %d; b=%d, %d, %d\n",
                   a, local.a,
                   Parent (main, this) ->a,
                   b, local.b,
                   Parent (main, this) ->b);
      }
```

```
    EndLocalScope;
          local .bar ( );
          printf ("foo: a=%d, b=%d\n", a, b);
       }
    EndOuterScope;
    local.a = 1;
    local.b = 2;
    local.foo ( );
}
```

其输出是：

```
bar:a = 5, 3, 1 ; b = 6, 4, 2
foo:a = 1, b = 2
```

下面是块结构的视频游戏的部分实现代码。

D.4 块结构的视频游戏代码

```
# include <block.h>
# include <iostream.h>
# include <curses.h>

int main( )
{
  LocalScope(BallGame):
    void DrawSides( ) {
      for(int i = 0; i < 66; i + +) mvaddch(0, i, 's');
      for(i = 0;i < 13;i + +) {
        mvaddch(i, 0, 's');
        mvaddch(i, 65, 's');
      }
      leftWall = 0;
      rightWall = 65;
    }

    struct Ball {
      int yposition, xposition;
      int yspeed, xspeed;
      Ball( ) {
        yposition = 6;
        xposition = 33;
        yspeed = 1;
        xspeed = 1;
  }
} ball;
struct Bat {
  int yposition, position, length, xspeed;
  Bat( ) {position = 40;
    yposition = 11;
    length = 4;
    xspeed = 0;
  }
} bat;

void DrawBat( ) {
```

```
  for (iht i = bat .position;
                 i < bat.position + bat.length; i + +) {
    mvaddch(bat.yposition, i, 'b');
  }
}

void EraseBat( ) {
  for (int i = bat .position;
                 i < bat.position + bat.length; i + +) {
    mvaddch(bat.yposition, i, ' ');
  }
}

short leftWall, rightWall;

int score;

void DrawWall ( ) {
  for (int i = 0; i < 66; i + +) {
    mvaddch (3, i, 'w');
    mvaddch (4, i, 'w');
    mvaddch (5, i, 'w');
  }
}

char ballIsInPlay( ) {
  return ball.xposition < rightWall &&
    ball.xposition > leftWall &&
    ball.yposition < = bat.yposition;
}

    void PlayGame( ) {
      LocalScope(PlayGame):
        char key;     //从键盘

        short ballsLeft;
        void PlayABall( ) {
          LocalScope(PlayABall):
            void CheckBallPosition( ) {
              PlayGameLocalScope *p =
                                Parent (PlayGame, this);
              BallGameLocalScope *p2 =
                               Parent(BallGame, p);
              int &y = p2 ->ball.yposition;
              int &x = p2 ->ball.xposition;
              if (x < = p2 ->leftWall +1 ||
                                x > = p2 -> rightWall - 1) {
                p2 ->ball.xspeed = -p2 ->ball.xspeed;
              }
              if (y < = 0) {
                p2 ->ball.yspeed = -p2 ->ball.yspeed;
              }
              char c = mvinch(y +p2 -> ball.yspeed,
                               x + p2 ->ball.xspeed);
              switch (c) {
```

```
            case 'w':
              mvaddch (y + p2 ->ball.yspeed,
                              x + p2 ->ball.xspeed, ' ')
              p2 -> score + +;
            case 'b':
        p2 -> ball.yspeed = -p2 ->ball.yspeed;
        p2 -> ball.xspeed = ->bat .xspeed;
        break;
    }
}

  void MoveBall( ) {
      PlayGameLocalScope *p =
                        Parent (PlayGame, this);
      BallGameLocalScope *p2 =
                        Parent (BallGame, p);
      mvaddch(p2 ->ball.yposition,
                        p2 ->ball.xposition ' ');
      p2 ->ball.xposition + = p2 ->ball.xspeed;
      p2 ->ball.yposition + = p2 ->ball.yspeed;
      mvaddch(p2 -> ball.yposition,
                        p2 ->ball.xposition, 'O');
}

void MoveBat(char key) {
    LocalScope(MoveBat):
      void MoveLeft(Bat& bat) {
        PlayABallLocalScope *p =
                  Parent(PlayABall, this);
        PlayGameLocalScope *p2 =
                  Parent(PlayGame, p);
        Parent(BallGame, p2) ->EraseBat( );
        if (bat.position >
          Parent (BallGame, p2) ->leftWall) {
            bat.position- -;
        }
        Parent(BallGame, p2) ->DrawBat( );
    }

      void MoveRight (Bat& bat) {
        PlayABallLocalScope *p =
                   Parent(PlayABall, this);
        PlayGameLocalScope *p2 =
                   Parent(PlayGame, p);
        Parent(BallGame, p2) ->EraseBat( );
        if (bat.position <
          Parent(BallGame, p2) ->rightWall _
          bat.length) {
          bat.position + +;
        }
        Parent(BallGame, p2) ->DrawBat( );
    }
 EndLocalScope;
            PlayGameLocalScope *p =
                        Parent(PlayGame, this);
            switch (key) {
```

```
            case'l':

            local.MoveLeft(
                        Parent(BallGame, p) ->bat);
            Parent(BallGame, p) ->bat.xspeed = -1;
            break;
          case 'r':
            local.MoveRight(
                         Parent(BallGame, p) ->bat);
            Parent(BallGame, p) ->bat .xspeed = 1;
            break;
          default:
            break;
          }
        }
      EndLocalScope;

      while (Parent(BallGame, this) ->ballIsInPlay( )) {
        local.CheckBallPosition( );
        local.MoveBall( );
        refresh( );
        key = getch( );
        local.MoveBat(key);
      }
    }
  EndLocalScope;

  local.ballsLeft = 4;
  while(local.ballsLeft > 0) {
    refresh( );
    local.key = getch( );
    local.PlayABall( );
    - - local.ballsLeft;
  }
 }
EndOuterScope;

int bestScore = 0;

initscr( );
cbreak( );
noecho( );

local.DrawSides( );
local.DrawBat( );

for (;;) {
    local.score = 0;
    local.DrawWall( );
    local.PlayGame( );
    if (local.score > bestScore) {
        bestScore = local.score;
    }
    cout << "best score is" << bestScore << "\n";
  }
}
```