C++ 编程思想

目　　录

第1章　对象的演化

计算机革命起源于一台机器，程序设计语言也源于一台机器。

然而计算机并不仅仅是一台机器，它是心智放大器和另一种有表述能力的媒体。这一点使它不很像机器，而更像我们大脑的一部分，更像其他有表述能力的手段，例如写作、绘画、雕刻、动画制作或电影制作。面向对象的程序设计是计算机向有表述能力的媒体发展中的一部分。

本章将介绍面向对象程序设计（OOP）的基本概念，然后讨论OOP开发方法，最后介绍使程序员、项目和公司使用面向对象程序设计方法而采用的策略。

本章是一些背景材料，如果读者急于学习这门语言的具体内容，可以跳到第 2章，然后再回过头来学习本章。

1.1　基本概念

C++包含了比面向对象程序设计基本概念更多的内容，读者应当在学习设计和开发程序之前先理解该语言所包含的基本概念。

1.1.1　对象：特性+行为[1]

第一个面向对象的程序设计语言是60年代开发的Simula-67。其目的是为了解决模拟问题。典型的模拟问题是银行出纳业务，包括出纳部门、顾客、业务、货币的单位等大量的“对象”。把那些在程序执行期间除了状态之外其他方面都一样的对象归在一起，构成对象的“类”，这就是“类”一词的来源。

类描述了一组有相同特性（数据元素）和相同行为（函数）的对象。类实际上就是数据类型，例如，浮点数也有一组特性和行为。区别在于程序员定义类是为了与具体问题相适应，而不是被迫使用已存在的数据类型。这些已存在的数据类型的设计动机仅仅是为了描述机器的存储单元。程序员可以通过增添他所需要的新数据类型来扩展这个程序设计语言。该程序设计系统欢迎创建、关注新的类，对它们进行与内部类型一样的类型检查。

这种方法并不限于去模拟具体问题。尽管不是所有的人都同意，但大部分人相信，任何程序都模拟所设计系统。OOP技术能很容易地将大量问题归纳成为一个简单的解，这一发现产生了大量的OOP语言，其中最著名的是Smalltalk——C++ 之前最成功的OOP语言。

抽象数据类型的创建是面向对象程序设计中的一个基本概念。抽象数据类型几乎能像内部类型一样准确工作。程序员可以创建类型的变量（在面向对象程序设计中称为“对象”或“实例”）并操纵这些变量（称为发送“消息”或“请求”，对象根据发来的消息知道需要做什么事情）。

1.1.2　继承：类型关系

类型不仅仅说明一组对象上的约束，还说明与其他类型之间的关系。两个类型可以有共同的特性和行为，但是，一个类型可能包括比另一个类型更多的特性，也可以处理更多的消息

[1] 这一描述部分引自我对《The Tao of Objects》（Gary Entsminger著）一书的介绍。

(或对消息进行不同的处理)。继承表示了基本类型和派生类型之间的相似性。一个基本类型具有所有由它派生出来的类型所共有的特性和行为。程序员创建一个基本类型以描述系统中一些对象的思想核心。由这个基本类型派生出其他类型，表达了认识该核心的不同途径。

例如，垃圾再生机要对垃圾进行分类。这里基本类型是“垃圾”，每件垃圾有重量、价值等等，并且可以被破碎、融化或分解。这样，可以派生出更特殊的垃圾类型，它们可以有另外的特性（瓶子有颜色）或行为（铝可以被压碎，钢可以被磁化）。另外，有些行为可以不同(纸的价值取决于它的种类和状态)。程序员可以用继承建立类的层次结构，在该层次结构中用类型术语来表述他需要解决的问题。

第二个例子是经典的形体问题，可以用于计算机辅助设计系统或游戏模拟中。这里基本类型是“形体”，每个形体有大小、颜色、位置等。每个形体能被绘制、擦除、移动、着色等。由此，可以派生出特殊类型的形体：圆、正方形、三角形等，它们中的每一个都有另外的特性和行为，例如，某些形体可以翻转。有些行为可以不同（计算形体的面积）。类型层次结构既体现了形体间的类似，又体现了它们之间的区别。

用与问题相同的术语描述问题的解是非常有益的，这样，从问题描述到解的描述之间就不需要很多中间模型（程序语言解决大型问题，就需要中间模型）。面向对象之前的语言，描述问题的解不可避免地要用计算机术语。使用对象术语，类型层次结构是主要模型，所以可以从现实世界中的系统描述直接进入代码中的系统描述。实际上，使用面向对象设计，人们的困难之一是从开始到结束过于简单。一个已经习惯于寻找复杂解的、训练有素的头脑，往往会被问题的简单性难住。

1.1.3 多态性

当处理类型层次结构时，程序员常常希望不把对象看作是某一特殊类型的成员，而把它看作基本类型成员，这样就可以编写不依赖于特殊类型的代码。在形体例子中，函数可以对一般形体进行操作，而不关心它们是圆、正方形还是三角形。所有的形体都能被绘制、擦除和移动，所以这些函数能简单地发送消息给一个形体对象，而不考虑这个对象如何处理这个消息。

这样，新添类型不影响原来的代码，这是扩展面向对象程序以处理新情况的最普通的方法。例如，可以派生出形体的一个新的子类，称为五边形，而不必修改那些处理一般形体的函数。通过派生新子类，很容易扩展程序，这个能力很重要，因为它极大地减少了软件维护的花费。(所谓“软件危机”正是由软件的实际花费远远超出人们的想象而产生的。)

如果试图把派生类型的对象看作它们的基本类型（圆看作形体，自行车看作车辆，鸬鹚看作鸟），就有一个问题：如果一个函数告诉一个一般形体去绘制它自己，或者告诉一个一般的车辆去行驶，或者告诉一只一般的鸟去飞，则编译器在编译时就不能确切地知道应当执行哪段代码。同样的问题是，消息发送时，程序员并不想知道将执行哪段代码。绘图函数能等同地应用于圆、正方形或三角形，对象根据它的特殊类型来执行合适的代码。如果增加一个新的子类，不用修改函数调用，就可以执行不同的代码。编译器不能确切地知道执行哪段代码，那么它应该怎么办呢？

在面向对象的程序设计中，答案是巧妙的。编译器并不做传统意义上的函数调用。由非OOP编译器产生的函数调用会引起与被调用代码的“早捆绑”，对于这一术语，读者可能还没有听说过，因为从来没有想到过它。早捆绑意味着编译器对特定的函数名产生调用，而连接器确定调用执行代码的绝对地址。对于 OOP，在程序运行之前，编译器不确定执行代码的地址，所以，当消息发送给一般对象时，需要采用其他的方案。

为了解决这一问题，面向对象语言采用“晚捆绑”的思想。当给对象发送消息时，在程序

运行之前不去确定被调用的代码。编译器保证这个被调用的函数存在，并完成参数和返回值的类型检查，但是它不知道将执行的准确代码。

为了实现晚捆绑，编译器在真正调用的地方插入一段特殊的二进制代码。通过使用存放在对象自身中的信息，这段代码在运行时计算被调用函数的地址（这一问题将在第 14章中详细介绍）。这样，每个对象就能根据一个指针的内容有不同的行为。当一个对象接收到消息时，它根据这个消息判断应当做什么。

程序员可以用关键字 virtual表明他希望某个函数有晚捆绑的灵活性，而并不需要懂得virtual的使用机制。没有它，就不能用 C++做面向对象的程序设计。 Virtual函数（虚函数）表示允许在相同家族中的类有不同的行为。这些不同是引起多态行为的原因。

1.1.4 操作概念：OOP程序像什么

我们已经知道，用C 语言编写的过程程序就是一些数据定义和函数调用。要理解这种程序的含义，程序员必须掌握函数调用和函数实现的本身。这就是过程程序需要中间表示的原因。中间表示容易引起混淆，因为中间表示的表述是原始的，更偏向于计算机，而不偏向于所解决的问题。

因为 C++ 向 C 语言增加了许多新概念，所以程序员很自然地认为， C++程序中的main()会比功能相同的 C 程序更复杂。但令人吃惊的是，一个写得很好的C++程序一般要比功能相同的C程序更简单和容易理解。程序员只会看到一些描述问题空间对象的定义（而不是计算机的描述），发送给这些对象的消息。这些消息表示了在这个空间的活动。面向对象程序设计的优点之一是通过阅读，很容易理解代码。通常，面向对象程序需要较少的代码，因为问题中的许多部分都可以用已存在的库代码。

1.2 为什么C++会成功

C++能够如此成功，部分原因是它的目标不只是为了将 C语言转变成OOP语言（虽然这是最初的目的），而且还为了解决当今程序员，特别是那些在 C语言中已经大量投资的程序员所面临的许多问题。人们已经对OOP语言有了这样传统的看法：程序员应当抛弃所知道的每件事情并且从一组新概念和新文法重新开始，他应当相信，最好丢掉所有来自过程语言的老行装。从长远角度看，这是对的。但从短期角度看，这些行装还是有价值的。最有价值的可能不是那些已存在的代码库（给出合适的工具，可以转变它），而是已存在的头脑库。作为一个职业 C程序员，如果让他丢掉他知道的关于 C的每一件事，以适应新的语言，那么，几个月内，他将毫无成果，直到他的头脑适应了这一新范例为止。如果他能调整已有的 C知识，并在这个基础上扩展，那么他就可以继续保持高效率，带着已有的知识，进入面向对象程序设计的世界。因为每个人有他自己的程序设计模型，所以这个转变是很混乱的。因此， C++成功的原因是经济上的：转变到OOP需要代价，而转变到C++所花的代价较小。

C++的目的是提高效率。效率取决于很多东西，而语言是为了尽可能地帮助使用者，尽可能不用武断的规则或特殊的性能妨碍使用者。 C++成功是因为它立足于实际：尽可能地为程序员提供最大便利。

1.2.1 较好的C

即便程序员在C++环境下继续写C代码，也能直接得到好处，因为C++堵塞了C语言中的一

些漏洞，并提供更好的类型检查和编译时的分析。程序员必须先说明函数，使编译器能检查它们的使用情况。预处理器虚拟删除值替换和宏，这就减少了查找疵点的困难。C++有一个性能，称为references(引用)，它允许对函数参数和返回值的地址进行更方便的处理。函数重载改进了对名字的处理，使程序员能对不同的函数使用相同的名字。另外，名字空间也加强了名字的控制。许多性能使C的更安全。

1.2.2 采用渐进的学习方式

与学习新语言有关的问题是效率的问题。所有公司都不可避免地因软件工程师学习新语言而突然降低了效率。C++是对C的扩充，而不是新的文法和新的程序设计模型。程序员学习和理解这些性能，逐渐应用并继续创建有用的代码。这是C++成功的最重要的原因之一。

另外，已有的C代码在C++中仍然是有用的，但因为C++编译器更严格，所以，重新编译这些代码时，常常会发现隐藏的错误。

1.2.3 运行效率

有时，以程序执行速度换取程序员的效率是值得的。假如一个金融模型仅在短期内有用，那么快速创建这个模型比所写程序能更快速执行重要。很多应用程序都要求有一定的运行效率，所以C++在更高运行效率时总是犯错。C程序员非常重视运行效率，这让他们认为这个语言不太庞大，也不太慢。产生的代码运行效率不够时，程序员可以用C++的一些性能做一些调整。

C++不仅有与C相同的基本控制能力（和C++程序中直接写汇编语言的能力），非正式的证据指出，面向对象的C++程序的速度与用C写的程序速度相差在±10%之内，而且常常更接近。用OOP方法设计的程序可能比C的对应版本更有效。

1.2.4 系统更容易表达和理解

为适合于某问题而设计的类当然能更好地表达这个问题。这意味着写代码时，程序员是在用问题空间的术语描述问题的解（例如“把锁链放在箱子里”），而不是用计算机的术语，也就是解空间的术语，描述问题的解（例如“设置芯片的一位即合上继电器”）。程序员所涉及的是较高层的概念，一行代码能做更多的事情。

易于表达所带来的另一个好处是易于维护。据报道，在程序的整个生命周期中，维护占了花费的很大部分。如果程序容易理解，那么它就更容易维护，还能减少创建和维护文档的花费。

1.2.5 “库”使你事半功倍

创建程序的最快方法是使用已经写好的代码：库。C++的主要目标是让程序员能更容易地使用库，这是通过将库转换为新数据类型（类）来完成的。引入一个库，就是向该语言增加一个新类型。编译器负责这个库如何使用，保证适当的初始化和清除，保证函数被正确地调用，因此程序员的精力可以集中在他想要这个库做什么，而不是如何做上。

因为程序的各部分之间名字是隔离的，所以程序员想用多少库就用多少库，不会有像C语言那样的名字冲突。

• 模板的源代码重用

一些重要的类型要求修改源代码以便有效地重用。模板可以自动完成对代码的修改，因而

是重用库代码特别有用的工具。用模板设计的类型很容易与其他类型一起工作。因为模板对程序员隐藏了这类代码重用的复杂性，所以特别好用。

1.2.6 错误处理

在C语言中，错误处理声名狼藉。程序员常常忽视它们，对它们束手无策。如果正在建大而复杂的程序，没有什么比让错误隐藏在某处，且不能指出它来自何处更糟的了。 C++的异常处理（见第17章的内容）保证能检查到错误并进行处理。

1.2.7 大程序设计

许多传统语言对程序的规模和复杂性有自身的限制。例如， BASIC对于某些类型的问题能很快解决，但是如果这个程序有几页纸长，或者超出该语言的正常解题范围，那么它可能永远算不出结果。C语言同样有这样的限制，例如当程序超过 50 000行时，名字冲突就开始成为问题。简言之，程序员用光了函数和变量名。另一个特别糟糕的问题是如果 C语言中存在一些小漏洞 —— 错误藏在大程序中，要找出它们是极其困难的。

没有清楚的文字告诉程序员，什么时候他的语言会失效，即便有，他也会忽视它们。他不说“ 我的BASIC程序太大，我必须用 C重写 ”，而是试图硬塞进另外几行，增加额外的性能。所以额外的花费就悄悄增加了。

设计C++的目的是为了辅助大程序设计，也就是说，去掉小程序和大程序之间复杂性的分界。当程序员写hello-world类实用程序时，他确实不需要用OOP、模板、名字空间和异常处理，但当他需要的时候，这些性能就有用了。而且，编译器在排除错误方面，对于小程序和大程序一样有效。

1.3 方法学介绍

所谓方法学是指一组过程和启发式，用以减少程序设计问题的复杂性。在 OOP中，方法学是一个有许多实践的领域。因此，在程序员考虑采用某一方法之前，了解该方法将要解决的问题是很重要的。对于C++，有一点是确实的：它本身就是希望减少程序表达的复杂性。从而不必用更复杂方法学。对于用过程语言的简单方法所不能处理的大型问题，在 C++中用一些简单的方法就足够了。

认识到“ 方法学 ”一词含义太广是很重要的。实际上，设计和编写程序时，无论做什么都在使用一种方法。只不过因为它是程序员自己的方法而没有意识到。但是，它是程序员编程中的一个过程。如果过程是有效的，只需要用 C++做很小的调整。如果程序员对他的效率和调整程序的方法不满意，他可以考虑采用一种更正式的方法。

1.3.1 复杂性

为了分析复杂性，先假设：程序设计制定原则来对付复杂性。

原则以两种方式出现，每种方式都被单独检查。

1) 内部原则体现在程序自身的结构中，机灵而有见解的程序员可以通过程序设计语言的表达方式了解这种内部原则。

2) 外部原则体现在程序的源信息中，一般被描述为“ 设计文档 ”（不要与产品文档混淆）。

我认为，这两种形式的原则互相不一致：一个是程序的本质，是为了让程序能工作而产生

的，另一个是程序的分析，为了将来理解和维护程序而产生的。创建和维护都是程序生命期的基本组成部分。有用的程序设计方法把两者综合为最合适的方式，而不偏向任何一方。

1.3.2 内部原则

程序设计的演化（C++只是其中的一步）从程序设计模型强加于内部开始，也就是允许程序员为内存位置和机器指令取别名。这是数字机器程序设计的一次飞跃，带动了其他方面的发展，包括从初级机器中抽象出来，向更方便地解决手边问题的模型发展。不是所有这些发展都能流行，起源于学术界并延伸进计算机世界的思想常常依赖于所适应的问题。

命名子程序的创建和支持子程序库的连接技术在50年代向前飞跃发展，并且孕育出了两个语言，它们在当时产生了巨大冲击，这就是为科学工作者使用的 FORTRAN（FORmula-TRANslation）和为商业者使用的COBOL（COmmon Business-Oriented Language）。纯计算机科学中很成功的语言是Lisp（List-Processing），而面向数学的语言应当是APL（A Programming Language）。

这些语言的共同特点是对过程的使用。 Lisp和APL的创造专注于语言的高雅 —— 语言的“ mission语句 ”嵌入在处理所有任务情况的引擎中。 FORTRAN和COBOL的创造是为了解决专门的问题，当这些问题变得更复杂，有新的问题出现时，它们又得到了发展。甚至它们进入衰退期后，仍在发展：FORTRAN和COBOL的版本都面向对象进行了扩充（后时髦哲学的基本原则是：任何具有自己独特生活方式的组织，其主要目标就是使这种生活方式永存）。

命名子程序在程序设计中起了重要作用，语言的设计围绕着这一原则，特别是 Algol和Pascal。同时另外一些语言也出现了，它们成功地解决了程序设计的一些子集问题，并将它们有序排列。最有趣的两个语言是Prolog和FORTH。前者是围绕着推理机而建立的（在其他语言中常常称作库）。后者是一个可扩充语言，允许程序员重新形成这个语言，以适应所解决的问题，观念上类似于面向对象程序设计。 FORTH还可以改变语言，因而很难维护，并且是内部原则概念最纯正的表达，它强调的是问题一时的解，而不是对这个解的维护。

人们还创造了其他许多语言，以解决某一部分的程序设计问题。通常，这些语言以特定的目标开始。例如，BASIC（Beginners All-purpose Symbolic Instruction Code）是在60年代设计的，目的是使程序设计对初学者更简单。 APL的设计是为了数学处理。两种语言都能够解决其他问题，而关键在于它们是否是这些问题集合最理想的解。有一句笑话是，“ 带着锤子三年，看什么都是钉子 ”。这反映了根本的经济学真理：如果我们只有 BASIC或APL语言，特别是，当最终期限很短且这个解的生命期有限时，它就是我们问题最好的解。

然而，最终考虑两个因素：复杂性的管理和维护（将在下一部分讨论）。即这种语言首先是为某一领域开发的，而程序员又不愿花很长时间来熟悉这门语言，其结果只能使程序越来越长，使手头的问题屈服于语言。界限是模糊的：谁能说什么时候您的语言会使您失望呢？这不是马上就出现的。

问题的解开始变长，并且对于程序员更具挑战性。为了知道语言大概的限制，你得更聪明，这种聪明变成了一种标准，也就是“ 为了使该语言工作而努力 ”。这似乎是人类的操作方式，而不是遇到缺陷就抱怨，并且不再称它为缺陷。

最终，程序设计问题对于求解和维护变得太困难了，即求得的解太昂贵了。人们最终明白了，程序的复杂性超出了我们能够处理的程度。尽管一大类程序设计要求开发期间去做大部分工作并创建要求最小维护的解（或者简单地丢掉这个解，或者用不同的解替换它），但这只是问题的一部分。一般情况是，我们把软件看作是为人们提供服务的工具。如果用户的需要变化

了，服务就必须随着变化。这样，当第一版本开始运行时，项目并没有结束。项目是一个不断进化的生命体。程序的更新变成了一般程序设计问题的一个部分。

1.3.3 外部原则

为了更新和改善程序，需要更新思考问题的方法。它不只是“我们如何让程序工作”，而是“我们如何让程序工作并且使它容易改变”。这里就有一个新问题：当我们只是试图让程序工作时，我们可以假设开发组是稳定的（总之，我们可以希望这样），但是，如果我们正在考虑程序的整个生命期，就必须假设开发组成员会改变。这意味着，新组员必须以某种方式学习原程序的要点，并与老组员互相通讯（也许通过对话）。这样，该程序就需要某种形式的设计文档。

因为只想让程序工作，文档并不是必需的，所以还没有像由程序设计语言强加于程序那样的、强加于创建文档的规则。这样，如果要求文档满足特定的需要，就必须对文档强加外部原则。文档是否“工作”，这很难确定（并且需要在程序一生中验证），因此，对外部原则“最好”形式的争论.比对“最好”程序设计语言的争论更激烈。

决定外部原则时，头脑中的重要问题是“我准备解决什么问题”。问题的根本就是上面所说的“我们如何让它工作和使它容易改变”。然而，这个问题常常有多种解释：它变成了“我如何才能与FoobleBlah文档规范说明一致，以使政府会为此给我拨款”。这样，外部原则的目的是为了建立文档，而不是为了设计好的、可维护的程序。文档竟然变得比程序本身更重要了。

被问到未来一般和特殊的计算的方向时，我会从这样的问题开始：哪种解花费较少？假设这个解满足需要，价格的不同足以使程序员放弃他当前做事情的习惯方式吗？如果他的方法包括存储在项目分析和设计过程中所创建的每个文档，并且包括当项目进化时维护这些文档，那么当项目更新时，他的系统将花费很大，但是它能使新组员容易理解（假设没有那么多的使人害怕阅读的文档）。这样创建和维护方法的花费会和它打算替代方法的花费一样多。

外部结构系列的另一个极端是最小化方法。为完成设计而进行足够的分析，然后丢掉它们，使得程序员不再花时间和钱去维护它；为开始编码而做足够的设计，然后丢掉这个设计，使得程序员不再花时间和钱去维护这些文档；然后使得代码是一流的和清晰的，代码中只需要最少的注释。为了使新组员快速参与项目，代码连同注释就足够了。因为在所有这些乏味的文档上，新组员只需花费很少的时间（总之，没有人真地理解它们），所以他能较快地参与工作。

即便不维护文档，丢掉文档也不是最好的办法，因为这毕竟是程序员所做的有效工作。某些形式的文档通常是必须的（参看本章后面的描述）。

1. 通讯

对于较大的项目，期望代码像文档一样充分是不合理的，尽管我们在实际中常常这样期望。但是，代码包含了我们实际上希望外部原则所产生的事物的本质：通讯。我们只是希望能与改进这个程序的新组员通讯就足够了。但是，我们还想使花费在外部原则上的钱最少，因为最终人们只为这个程序所提供的服务付钱，而不是为它后面的设计文档付钱。为了真正有用，外部原则应当做比只产生文档更多的事情——它应当是项目组成员在创建设计时为了讨论问题而采用的通讯方法。理想的外部原则目标是使关于程序分析和设计的通讯更容易。这对于现在为这个程序而工作的人们和将来为这个程序而工作的人们是有帮助的。中心问题不只是为了能通讯，而为了产生好的设计。

人们（特别是程序员）被计算机吸引（由于机器为他们做工作），出于经济原因，要求开

发者为机器做大量工作的外部原则似乎从一开始就注定要失败。成功的方法（也就是人们习惯的方法）有两个重要的特征：

1) 它帮助人们进行分析和设计。这就是，用这种方法比用别的方法对分析和设计中的思考和通讯要容易得多。目前的效率和采用这种方法后的效率应当明显不同。否则，人们可能还留在原地。还有，它的使用必须足够简单，不需用手册。当程序员正在解决问题时，要考虑简单性，而不管他适用于符号还是技术。

2) 没有短期回报，就不会加强投资。在通向目标的可见的进展中，没有短期回报，人们就不会感到采用一种方法能使效率提高，就会回避它。不能把这个进展误认为是从一种中间形式到另一种中间形式的变换。程序员可以看到他的类，连同类之间互相发送的消息一起出现。为某人创造一种方法，就像武断的约束，因为它是简单的心理状态：人们希望感到他们正在做创造性的工作，如果某种方法妨碍他们而不是帮助他们飞快地接近目标，他们将设法绕过这种方法。

2. 量级

在方法学上反对我的观点之一是：“好了，您能够侥幸成功是因为您正在做的小项目很短。”听众对“小”的理解因人而异。虽然这种看法并不全对，但它包含一个正确的核心：我们所需要的原则与我们正在努力解决问题的量级有关。小项目完全不需要外部原则，这不同于个别程序员正在解的生命期问题的模式。涉及很多人的大项目会使人们之间有一些通讯，所以必须使通讯具有形式化方法，以使通讯有效和准确。

麻烦的是介于它们之间的项目。它们对外部原则的需要程度可能在很大程度上依赖于项目的复杂性和开发者的经验。确实，所有中等规模的项目都不需要忠实于成熟的方法，即产生许多报告和很多文档。一些项目也许这样做，但许多项目可以侥幸成功于“方法学简化”（代码多而文档少）。我们面前的所有方法学的复杂性可以减少到 80%～20%的（或更少的）规则。我们正在被方法学的细节淹没，在所解决的程序设计问题中，可能只有不足 20%的问题需要这些方法学。如果我们的设计是充分的，并且维护也不可怕，那么我们也许不需要方法学或不全部需要它。

3. OOP是结构化的吗

现在提出一个更有意义的问题。为了使通讯方便，假设方法学是需要的。这种关于程序的元通讯是必须的，因为程序设计语言是不充分的——它太原始，趋向于机器范例，对于谈论问题不很有用。例如，过程程序设计方法要求用数据和变换数据的函数作为术语谈论程序。因为这不是我们讨论实际问题的方法，所以必须在问题描述和解描述之间翻译来翻译去。一旦得到了一个解描述并且实现了它，以后无论何时对这个解描述做改变就要对问题描述做改变。这意味着必须从机器模式返回问题空间。为了得到真正可维护的程序，并且能够适应问题空间上的改变，这种翻译是必须的。投资和组织的需要似乎要求某种外部原则。过程程序的最重要的方法学是结构化技术。

现在考虑，是否解空间上的语言可以完全脱离机器模式？是否可以强迫解空间使用与问题空间相同的术语？

例如，在气候控制大楼中的空气调节器就变成了气候调节程序的空气调节器，自动调温器变成了自动调温程序，等等。（这是按直觉做的，与OOP不一致）。突然，从问题空间到解空间的翻译变成了次要问题。可以想象，在程序分析、设计和实现的每一阶段，能使用相同的术语学、相同的描述，这样，这个问题就变成了“如果文档（程序）能够充分地描述它自身，我们仍然需要关于这个文档的文档吗？”如果OOP做它所主张的事情，程序设计的形式就变成了这

样：在结构化技术中所遇到的困难在新领域中可能不复存在了。

这个论点也为一个思想实验所揭示。假设程序员需要写一些小实用程序，例如能在文本文件上完成一个操作的程序（就像在第6章的后几页上可找到的那样），它们要程序员花费几分钟，最困难的要花费几小时去写。现在假设回到50年代，这个项目必须用机器语言或汇编语言来写，使用最少的库函数，它需要许多人几星期或几个月的时间。在 50年代需要大量的外部原则和管理，现在不需要了。显然，工具的发展已经极大地增加了我们不用外部原则解决问题的复杂性（同样很显然，我们将发现的问题也更加复杂）。

这并不是说可以不需要外部原则，有用的OOP外部原则解决的问题与有用的过程程序设计外部原则所解决的问题不同，特别是，OOP方法的目标首先必须是产生好的设计。好设计不仅要促进重用，而且它与项目的各级开发者的需要是一致的。这样，开发者就会更喜欢采用这样的系统。让我们基于这些观点考虑OOP设计方法中的一些问题。

1.3.4 对象设计的五个阶段

对象的设计不限于写程序的时期，它出现在一系列阶段。有这种观点很有好处，因为我们不再期望设计立刻尽善尽美，而是认识到，对对象做什么和它应当像什么的理解是随着时间的推移而产生的。这个观点也适用于不同类型程序的设计。特殊类型程序的模式是通过一次又一次地求解问题而形成的[1]。同样，对象有自己的模式，通过理解、使用和重用而形成。

下面是描述，不是方法。它简直就是对象期望的设计出现时的观察结果。

1) 对象发现　这个阶段出现在程序的最初分析期间。可以通过寻找外部因素与界线、系统中的元素副本和最小概念单元而发现对象。如果已经有了一组类库，某些对象是很明显的。类之间的共同性（暗示了基类和继承类），可以立刻出现或在设计过程的后期出现。

2) 对象装配　我们在建立对象时会发现需要一些新成员，这些新成员在对象发现时期未出现过。对象的这种内部需要可能要用新类去支持它。

3) 系统构造　对对象的更多要求可能出现在以后阶段。随着不断的学习，我们会改进我们的对象。与系统中其它对象通讯和互相连接的需要，可能改变已有的类或要求新类。

4) 系统扩充　当我们向系统增添新的性能时，可能发现我们先前的设计不容易支持系统扩充。这时，我们可以重新构造部分系统，并很可能要增加新类。

5) 对象重用　这是对类的真正的重点测试。如果某些人试图在全新的情况下重用它，他们会发现一些缺点。当我们修改一个类以适应更新的程序时，类的一般原则将变得更清楚，直到我们有了一个真正可重用的对象。

对象开发原则

在这些阶段中，提出考虑开发类时所需要的一些原则：

1) 让特殊问题生成一个类，然后在解其他问题时让这个类生长和成熟。

2) 记住，发现所需要的类，是设计系统的主要内容。如果已经有了那些类，这个项目就不困难了。

3) 不要强迫自己在一开始就知道每一件事情，应当不断地学习。

4) 开始编程，让一部分能够运行，这样就可以证明或反驳已生成的设计。不要害怕过程语言风格的细面条式的代码——类分割可以控制它们。坏的类不会破坏好的类。

5) 尽量保持简单。具有明显用途的不太清楚的对象比很复杂的接口好。我们总能够从小的

[1] 参看Design Patterns:Elements of Reusable Object-Oriented Software by Erich Gamma et al., Addison-Wesley, 1995。

和简单的类开始，当我们对它有了较好地理解时再扩展这个类接口，但不可能简化已存在的类接口。

1.3.5 方法承诺什么

由于不同的原因，方法承诺的东西往往比它们能够提供的东西多得多。这是不幸的，因为当策略和不实际的期望同时出现时程序员会疑神疑鬼。一些方法的坏名声，使得程序员丢弃了手上的其他方法，忽视了一些有价值的技术。

1. 管理者的银子弹

最坏的许诺是“这个方法将解决您的所有问题”。这一许诺也很可能用这样的思想表达，即一个方法将解决实际上不存在解的问题，或者至少在程序的设计领域内没有解的问题：一个贫穷的社团文化，疲惫的、互相疏远或敌对的项目组成员；不充分的时间和资源；或试图解决一个实际上不能解的问题（资源不足）。最好的方法学，不管它许诺什么，都不解决这些或类似的任何问题。无论OOP还是C++，都无助于这样的问题。不幸的是，在这种情况下管理员是这样的人：对于银子弹的警报[1]，他是最易动摇的。

2. 提高效率的工具

这就是方法应当成为的东西。提高生产效率不仅取决于管理容易和花费不大，而且取决于一开始就创建好的设计。由于一些方法学的创造动机是为了改善维护，所以它们就片面地强调维护问题，而忽视了设计的漂亮和完整。实际上，好的设计应当是首要的目标，好的 OOP设计也应当容易维护，但这是它的附加作用。

1.3.6 方法应当提供什么

不管为特殊方法提出什么要求，它都应当提供这一节所列出的基本功能：允许为讨论这个项目将完成什么和如何做而进行通讯的约定；支持项目结构化的系统；能用某抽象形式描述项目的一组工具（使得程序员能容易地观察和操作项目）。如过去介绍过的，一个更微妙的问题是该方法对待最宝贵资源——组员积极性的“态度”。

1. 通讯约定

对于很小的项目组，可以用紧密接触的方式自然维持通讯。这是理想的请况。 C++的最大的好处之一是它可以使项目由很少的项目组成员建立，因此，明白表示的通讯能使维护变得容易，因而通讯费用低，项目组能更快地建立。

情况并不总是这样理想，有可能项目组成员很多，项目很复杂，这就需要某种形式的通讯原则。方法提供一种在项目组成员之间形成“约定”的办法。可以用两种方式看待这样的约定：

1) 敌对的　约定基于参与的当事人之间互有疑问，以使得没有人出格且每个人都做应该做的事情。约定清楚地说明，如果他们不做这些事，会出现坏的结果。这样看待任何约定，我们就已经输了，因为我们已经认为其他人是不可信赖的了。如果不能信任某人，约定并不能确保好的行为。

2) 信息的　约定是一种努力，使每个人都知道我们已经在哪些方面取得了一致的意见。这是对通讯的辅助，使得每个人能看到它并说，“是的，这是我认为我们将要做的事情”。它是协议作出后对协议的表述，只是消除误解。这类约定能最小化，并容易读。

---

[1] A reference to vampires made in The Mythical Man-Month, by Fred Brooks, Addision-Wesley, 1975。

有用的方法不鼓励敌对的约定，而重点是在通讯上。

2. 使系统结构化

结构化是系统的核心。如果一个方法能做些事情，那么它就必须能够告诉程序员：

1) 需要什么类

2) 如何把它们连接在一起，形成一个工作系统。

一个方法产生这些回答需要一个过程，即首先对问题进行分析，最终对类、系统和类之间传递的消息进行某种表述。

3. 描述工具

模型不应当比它描述的系统更复杂。一种好的模型仅提供一种抽象。

程序员一定不会使用只对特殊方法有用的描述工具。他能使自己的工具适合自己的各种需要。（例如在本章后面，建议一种符号，用于商业字处理机。）下面是有用的符号原则：

1) 方法中不含有不需要的东西。记住，“七加减二”规则的复杂性。（瞬间，人们只能在头脑中存放这么多的条目。）额外的细节就变成了负担，必须维护它，为它花钱。

2) 通过深入到描述层，人们应当能够得到所需要的信息。即我们可以在较高的抽象层上创建一些隐藏的层，仅在需要时，它们才可见。

3) 符号应当尽可能少。“过多的噱头使得软件变坏”。

4) 系统设计和类设计是互相隔离的问题。类是可重用工具，而系统是对特殊问题的解（虽然系统设计也是可重用的）。符号应当首先集中在系统设计方面。

5) 类设计符号是必须的吗？由C++语言提供的类表达对大多数情况是足够的。如果符号在描述类方面不能比用OOP语言描述有一个明显的推进，那么就不要用它。

6) 符号应当在隐藏对象的内部实现。在设计期间，这些内部实现一般不重要。

7) 保持符号简单。我们想用我们的方法做的所有事情基本上就是发现对象及其如何互相连接以形成系统。如果一个方法或符号要求更多的东西，则应当问一问，该方法花费我们的时间是否合理。

4. 不要耗尽最重要的资源

我的朋友Michael Wilk来自学术界，也许并不具备做评判的资格（从某个人那里听说的新观点），但他观察到，项目、开发组或公司拥有的最重要的资源是积极性。不管问题如何困难，过去失败多么严重，工具多么原始或不成套，积极性都能克服这些障碍。

不幸的是，各种管理技术常常完全不考虑积极性，或因为不容易度量它，就认为它是“不重要”的因素，他们认为，如果管理完善，项目就能强制完成。这种认识有压制开发组积极性的作用，因为他们会感到公司除了利益动机以外就没有感兴趣的东西了。一旦发生这种现象，组员就变成了“雇员”，看着钟，想着感兴趣的、分心的事情。

方法和管理技术是建立在动机和积极性的基础上的，最宝贵的资源应当是对项目真正感兴趣。至少，应当考虑OOP设计方法对开发组士气起的作用。

5. “必”读

在选择任何方法之前，从并不想出售方法的人那儿得到意见是有帮助的。不真正地理解我们想要一种方法是为了做什么或它能为我们做什么，那么采用一种方法很容易。其他人正在用它，这似乎是很充足的理由。但是，人们有一种奇怪的心理：如果他们相信某件事能解决他们的问题，他们就将试用它（这是经验，是好的）。但是，如果它不能解决他们的问题，他们可能加倍地努力，并且开始大声宣布他们已经发现了伟大的东西（这是否定，是不好的）。这个假设是，如果见到同一条船上有其他人，就不感到孤单，即便这条船哪儿也不去。

这并不是说所有的方法学都什么也不做，而是用精神方法使程序员武装到牙齿，这些方法能使程序员保持实验精神（“它不工作，让我们再试其它方法”）和跳出否定方式（“不，这根本不是问题，每样东西都很好，我们不需要改变”）。我认为，在选择方法之前读下面的书，会为我们提供这些精神武器。

《软件的创造力》（Software Creativity，Robert Glass编，Prentice-Hall,1995）。这是我所看到的讨论整个方法学前景最好的书。它是由Glass已经写过的短文和文章以及收集到的东西组成的（P.J.Plauger是一个撰稿者），反映出他在该主题上多年的思考和研究。他们愉快地说明什么是必须的，并不东拉西扯和扫我们的兴，不说空话，而且，这里有几百篇参考文献。所有的程序员和管理者在陷入方法学泥潭之前应当读这本书[1]。

《人件》（Peopleware，Tom Demarco 和Timothy Lister 编，Dorset House,1987）。虽然他们有软件开发方面的背景，而且本书大体上是针对项目和开发组的，但是这本书的重点是人和他们的需要方面，而不是技术和技术的需要方面。他们谈论创造一个环境，在其中人们是幸福和高效率的，而不是决定人们应当遵守什么样的规则，以使得他们成为机器的合适部件。我认为，这本书后面的观点是对程序员在采用XYZ方法，然后平静地做他们总是做的事情时微笑和点头的最大贡献。

《复杂性》（Complexity, M. Mitchell Waldrop编，Simon & Schuster, 1992）。此书集中了Santa Fe, New Mexico的一组持不同观点的科学家，讨论单个原则不能解决的实际问题（经济学中的股票市场、生物学的生命原始形式、为什么人们做他们在社会中应做的事情，等等）。通过物理学、经济学、化学、数学、计算机科学、社会学和其他学科的交叉，对这些问题的一种多原则途径正在发展。更重要的是，思考这些极其复杂问题的不同方法正在形成。抛开数学的确定性，人们想写一个预言所有行为的方程，首先观察和寻找一个模式，用任何可能的手段模拟这个模式（例如，这本书编入了遗传算法）。我相信，这种思维是有用的，因为我们正在对管理越来越复杂软件项目的方法做科学观察。

1.4 起草：最小的方法

我首先声明，这一点没有证明过。我并不许诺——起草是起点，是其他思想的种子，是思想试验，尽管这是我在大量思考、适量阅读和在开发过程中对自己和其他人观察之后形成的看法。这是受我称之为“小说结构”的写作类的启示。“小说结构”出现在Robert McKee[2]的教学中，最初针对热心熟练的电影剧作家们，也针对小说家和编剧。后来我发现，程序员与这个人群有大量的共同点：他们的思想最终用某类文本形式表达，表达的结构能确定产品成功与否。有少量令人拍案称奇的上口小说，其他许多小说都很平庸，但有技巧，得到发行，大量不上口的小说得不到发表。当然，小说要描述，而程序要编写。

作家还有一些在程序设计中不太出现的约束：他们一般单独工作或可能在两个人的组中工作。这样，他们必须非常经济地使用他们的时间，放弃不能带来重要成果的方法。McKee的两个目标是将花费在电影编剧上的时间从一年减少到六个月，在这个过程中极大地提高电影编剧的质量。软件开发者可以有类似的目标。

使每个人都同意某些事情是项目启动过程中最艰苦的部分。系统的最小性应当能获得最独立的程序员的支持。

---

[1] 另外一本好“前景”的书是Object Lessons Tom Love著, SIGS Books, 1993。

[2] Through Two Arts, Inc.,12021 Wilshire Blvd. Suite 868, Los Angeles, CA 90025。

1.4.1 前提

该方法的描述是建立在两个重要前提的基础上的，在我们采用该思想的其他部分时必须仔细考虑这两个前提：

1) 与典型的过程语言（和大量已存在的语言）不同， C++语言和语言性能中有许多防护，程序员能建立自己的防护。这些防护意在防止程序员创建的程序破坏它的结构，无论在创建它的整个期间还是在程序维护期间。

2) 不管分析得如何透彻，这里还有一些有关系统的事直到设计时还没有揭示出来，更多的直到程序完成和运行时还没有揭示出来。因此，快速通过分析过程和设计过程以实现目标系统的测试是重要的。根据第一点，这比用过程语言更安全，因为C++中的防护有助于防止“面条”代码的创建。

着重强调第二点。由于历史原因，我们已经用了过程语言，因此在开始设计和实现之前开发组希望仔细地处理和了解每一微小的细节，这是值得表扬的。的确，当创建 DBMS时，彻底地了解消费者的需要是值得的。但是， DBMS是一类具有良好形式且容易理解的问题。在这一章中讨论的这类程序设计问题是wild-card变体问题，它不只简单地重新形成已知解，而是包括一个或多个wild-card因素——元素，在这里没有容易理解的先前的解，而必须进行研究[1]。在着手设计和实现之前彻底地分析wild-card问题会导致分析瘫痪，因为在分析阶段没有足够的信息解决这类问题。解这样的问题要求在整个周期中反复，要冒险（以产生感性认识，因为程序员正在做新事情，并且有较高的潜在回报）。结果是，危险由盲目“闯入”预备性实现而产生，但是它反而能减少在wild-card项目中的危险，因为人们能较早地发现一个特殊的设计是否可行。

这个方法的目标是通过建议解处理wild-card问题，得到最快的开发结果，使设计能尽早地被证明或反证。这些努力不会白费。这个方法常常建议，“建立一个解，然后再丢掉它”。用OOP，可能仍然丢掉一部分，但是因为代码被封装成类，不可避免地生产一些有用的类设计和第一次反复中发展一些对系统设计有价值的思想，它们不需要丢掉。这样，对某一问题快速地过一遍不仅产生对下一次分析、设计和实现的重复重要的信息，而且也为下一次的重复过程创建了代码基础。

这个方法的另一性能是能对项目早期部分的集体讨论提供支持。由于保持最初的文档小而简明，所以最初的文档可以由小组与动态创建该描述的领导通过几次集体讨论而创建，这不仅要求每个人的投入，而且还鼓励开发组中的每个人意见一致。也许更重要的是，它能在较高的积极性下完成一个项目（如先前注意到的，这是最基本的资源）。

表示法

作家的最有价值的计算机工具是字处理器，因为它容易支持文档的结构。对于程序设计项目，程序的结构通常是由某形式的分离的文档来表述的。因为项目变得更复杂，所以文档是必需的。这就出现了一类问题，如 Brooks[2] 所说：“数据处理的基本原则引出了试图用同步化方法维护独立文档的傻念头——而我们在程序设计文档方面的实践违反了我们自己的教义。我们典型的努力是维护程序的机器可读形式和一组独立可读的文档……。”

好的工具应当将代码和它的文档联系起来。

我们认为，使用熟悉的工具和思维模式是非常重要的， OOP的改变正面临着由它自己引起

[1] 我估计这样的项目的主要规则：如果多于一张 wild-card，则不计划它要费多长时间和它花费多少。这里有太多的自由度。

[2] The Mythical Man-Month, 出处同上。

的挑战。较早的OOP方法学已经因为使用精细的图形符号方案而受挫了。我们不可避免地要大量改变设计，因为我们必须改变设计以避免棘手的问题，所以用很难修改的符号表示设计是不好的。只是最近，才有处理这些图形符号的工具出现。容易使用设计符号的工具必须在希望人们使用这种方法之前就有了。把这种观点与在软件设计过程中要求文档这一事实结合，可以看出，最符合逻辑的工具是一个性能全面的字处理器[1]。事实上，每个公司都有了这些工具（所以试用这种方法不需要花费），大多数程序员熟悉它们，程序员习惯于用它们创建基本宏语言。这符合C++的精神，我们是建立在已有的知识和工具基础上的，而不是把它们丢掉。

这个方法所用的思维模式也符合这种精神。虽然图形符号在报告中表示设计是有用的[2]，但它不能紧密地支持集体讨论。但是，每个人都懂得画轮廓，而且很多字处理器有一些画轮廓的方法，允许抓取几块轮廓，很快地移动它们。这使交互集体讨论会上快速设计很完美。另外，人们可以扩展和推倒轮廓，决定于系统中粒度的不同层次。（如后描述）因为程序员创建了设计，创建了设计文档，所以关于项目状态的报告能用一个像运行编译器一样的过程产生。

1.4.2 高概念

建立的系统，无论如何复杂，都有一个基本的目的，它服务于哪一行业和它应满足什以基本需要。如果我们看看用户界面、硬件、系统特殊的细节、编码算法和效率问题，那么我们最终会发现它有简单和直接的核心。就像好莱坞电影中所谓的高概念，我们能用一两句话描述它。这种纯描述是起点。

高概念相当重要，因为它为我们的项目定调。它是委派语句，不需要一开始就正确（可以在完全清楚它之前完善这个论述或构思设计），它只是尝试，直到它正确。例如，在空中交通控制系统中，我们可以从系统的一个高概念开始，即准备建立“控制塔跟踪飞机。”但是当将该系统用于非常小的飞机场时，也许只有一个导航员或无人导航。更有用的模型不会使得正在创建的解像问题描述那么多：“飞机到达、卸货、服务和重新装货、离去。”

1.4.3 论述(treatment)

剧本的论述是用一两页纸写的故事概要，即高概念的外层。计算机系统发展高概念和论述的最好的途径可能是组成一个小组，该小组有一个具有写能力的辅助工具。在集体讨论中能提出建议，辅助工具在与小组相连的网络计算机上或在屏幕上表达这些思想。辅助工具只起捉刀人的作用，不评价这些思想，只是简单地使它们清楚和保持它们通顺。

论述变成了初始对象发现的起点和设计的第一个雏形，它也能在拥有辅助工具的小组内完成。

1.4.4 结构化

对于系统，结构是关键。没有结构，就会任意收集无意义事件。有了结构，就有了故事。故事的结构通过特征表示，特征对应于对象，情节结构对应于系统设计。

1. 组织系统

如前所述，对于这种方法，最主要的描述工具是具有概括功能的高级字处理器。

---

[1] 我的观察是基于我最熟悉的：Microsoft Word的扩展功能，它已被用于产生了这本书的照相机准备的页。

[2] 我鼓励这种选择，即用简单的方框、线和符号，它们在画字处理的包时是可用的，而不是很难产生的无定型的形状。

从“高概念”、“论述”、“对象”和“设计”的第一层开始。当对象被发现时它们就被放在第二层子段中，放在“对象”下面。增加对象接口作为第三层子段，放在对象的特殊类下面。如果基本表述文本产生，就在相应的子段下面作为标准文本。

因为这个技术包括键入和写大纲，不依靠图画，所以会议讨论过程不受创作该描述的速度限制。

2. 特征：发现初始对象

论述包括名词和动词。当我们列出它们后，一般将名词作为类，动词或者变为这些类的方法或者变为系统设计的进程。虽然在第一遍之后程序员可能对他找出的结构不满意，但请记住，这是一个反复的过程。在将来的阶段和后面的设计中可以增加另外的类和方法，因为那时程序员对问题会有更清晰的认识。这种构造方法的要点是不需要程序员当前完全理解问题，所以不期望设计一下子展现在程序员的面前。

从简单的论述检查开始，为每个已找出的唯一名称创建“对象”中的第二层子段。取那些很显然作用于对象的动词，置它们于相应名词下面的第三层方法子段。对每个方法增加参数表（即使它最初是空的）和返回类型。这就给程序员一个雏形以供讨论与完善。

如果一个类是从另一个类继承来的，则它的第二层子段应当尽可能靠近地放在这个基类之后，它的子段名应当显示这个继承关系，就像写代码：derived:public base时应当做的。这允许适当地产生代码。

虽然能设置系统去表示从公共接口继承来的方法，但目的是只创建类和它们的公共接口，其他元素都被认为是下面实现的部分，不是高层设计。如果要表示它们，它们应当作为相应类下面的文本层注解出现。

当决策点确定后，使用修改的 Occam’s Razor办法：考虑这个选择并选择最简单的一个，因为简单的类几乎总是最好的。向类增加元素很容易，但是随着时间的推移，丢掉元素就困难了。

如果需要培植这一过程，请看一个懒程序员的观点：您应当希望哪些对象魔术般地出现，用以解决您的问题？让一些可以用的类和各种系统设计模式作为手头的参考是有用的。

我们不要总是在对象段里，分析这个论述时应当在对象和系统设计之间来回运动。任何时候，我们都可能想在任何子段下面写一些普通文本，例如有关特殊类或方法的思想或注解。

3. 情节：初始系统设计

从高概念和论述开始，会出现一些子情节。通常，它们就像“输入、过程、输出”或“用户界面、活动”一样简单。在“设计”下面，每个子情节有它自己的第二层子段。大多数故事沿用一组公共情节。在oop中，这种类似被称为“模式”。查阅在oop设计模式上的资源，可以帮助对情节的搜索。

我们现在正在创建系统的粗略草图。在集体讨论会上，小组中的人们对他们认为应该出现在系统中的活动提出建议，并且分别记录，不需要为将它与整个系统相连而工作。让整个项目组，包括机械设计人员（如果需要）、市场人员、管理人员，都出席会议。这是特别重要的，这不仅使得每个人心情舒畅，因为他们的意见已经被考虑，而且每一个人的参加对会议都是有价值的。

子情节有一组变化的阶段或状态，有在阶段之间变化的条件，有包含在每个过渡中的活动。在特殊的子情节下面，每个阶段都给出它自己的第三层子段。条件和过渡被描写为文本，放在这个阶段的标题下。情况如果理想，我们最终能够写出每个子情节的基本的东西（因为设计是反复过程），作为对象的创建并向它们发送的消息。这就变成了这个子情节的最初代码。

设计发现和对象发现过程类似，因此我们可以在会议过程中将子条目增加到这两个段中。

1.4.5 开发

这是粗设计到编译代码的最初转换，编译代码能被测试，特别是，它将证明或者反证我们的设计。这不只是一遍处理，而是一系列写和重写的开始，所以，重点是从文档到代码的转换，该转换用这样一种方法，即通过对代码中的结构或有关文字的改变重新产生文档。这样，在编码开始后（和不可避免的改变出现后）产生设计文档就变得非常容易了，而设计文档能变成项目进展的报告工具。

1. 初始翻译

通过在第一层的标题中使用标准段名“对象”和“设计”，我们就能运行我们的工具以突出这些段，并由它们产生文件头。依据我们所在的主段和正在工作的子段层，我们将完成不同的工作。最容易的办法可能是让我们的工具或宏把文档分成小块并且相应地在每一小块上工作。

对于“对象”中的每个第二层段，在段名（类名和它的基类名，如果有的话）中会有足够的信息用以自动地产生类声明，在这个类名下面的每个第三层子段，段名（成员函数名、参数表和返回类型）中会有足够的信息用以产生这个成员函数的声明。我们的工具将简单地管理这些并创建类声明。

为了使问题简单，单个类声明将出现在每个头文件中。命名这些头文件的最好办法，也许是包括这个文件名，以作为这个类的第二层段名中的标记信息。

编制情节可以更精细。每个子情节可以产生一个独立的函数，由内部 main( )调用，或者就是main( )中的一段。从一个能完成我们的工作的事情开始，更好的模式可能在以后的反复中形成。

2. 代码产生

使用自动工具（大部分字处理工具对此是合适的）：

1) 为“对象”段中描述的每个类产生一个头文件，也就是为每一个类创建一个类声明，带有公共接口函数和与它们相联系的描述块；对每个类附上在以后很容易分析的专门标号。

2) 为每个子情节产生头文件，并且将它的描述拷贝在文件的开头，作为注释块，接下来跟随函数声明。

3) 用它的概要标题层标记每个子情节、类和方法，形成加标号的、被注释的标识符：//#[1]、// #[2]等等。所有产生的文件都有文档注释，放在带有标号的专门标识块中。类名和函数声明也保留注释标记。这样，转换工具能检查、提取所有信息，并用文档描述语言更好地重新产生源文档，例如用Rich Text Format(RTF)描述语言。

4) 接口和情节在这时应当是可编译的（但不可连接），因此可以做语法检查。这将保证设计的高层完整性。文档能由当前正在编译的文件重新产生。

5) 在这个阶段，有两件事会发生。如果设计是在早期，那么我们可能需要继续加工处理集体讨论会的文档（而不是代码）或小组负责的那部分文档。然而，如果设计是完全充足的，那么我们就可以开始编码。如果在编码阶段增加接口元素，则这些元素必须连同加过标号的注释一起由程序员加标号，所以重新产生的程序能用新信息产生文档。

如果我们拥有前端编译器，我们确实可以对类和函数自动进行编译，但是它是大作业，并且这个语言正在演化。使用明确的标号，是相当不安全的，商业浏览工具能用以检验是否所有的公共函数都已经形成文档了（也就是，它们已加标号了）。

1.4.6 重写

这类似于重写电影剧本以完善它，使它更好。在程序设计中，这是重复过程，我们的程序从好到更好，在第一遍中还没有真正理解的问题变得清楚了。我们的类从在单个项目中使用进化为可重用的资源。

从工具的观点看，转换该过程略微复杂，我们希望能分解头文件，使我们能重新整理这些文件使它们成为设计文档，包括在编码过程中已经做过的全部改变。在设计文档中对设计有任何改变，头文件也必须完全重建，这样就不会丢失为了得到在第一遍反复中编译所需要的头文件而做的任何工作。因此，我们的工具不仅应当能找出加标号的信息，将它们变成段层和文本层，而且还应当能发现其他信息，为其他信息加标号和存放其他信息，例如在每个文件开头的#include。我们应当记住，头文件表达了类设计而我们必须能够由设计文档重新产生头文件。

我们还应当注意文本层注解和讨论，它们最初产生时被转换为加标号的注释，比在设计演化过程中程序员修改的内容更多。基本上，这些注解和讨论被收集并放在各自的地方，因此设计文档反映了新的信息。这就允许我们去改变这些信息，并且返还到已产生的头文件中。

对于系统设计（main()和任何支持函数），我们也可能想获得整个文件，添加段标识符，例如A、B、C等等，作为加标号的注释（不用行号，因为行号会改变），并附上段描述（然后返还main()文件，作为加标号的文本）。

我们必须知道什么时候停止，什么时候重复设计。理想的情况是，我们达到了目标功能，处在完善和增加新功能的过程中，最后期限到来，强迫我们停止并发出我们的版本（记住，软件是预约业务）。

1.4.7 逻辑

我们会周期性地希望知道项目在什么地方需要重新整理文档。如果是在网上使用自动化工具，这个过程无关紧要。经常地整集和维护设计文档是项目领导者或管理者的责任，而项目组或个人只对文档的一小部分负责（也就是他们的代码和注释）。

对于补充功能，例如类图表，可以用第三方工具生成，并自动地添加到文档中。

在任何时候，都可以通过简单地“刷新”文档生成当前的报告。这样可以看到程序各部分的情况，也支援了项目组，并为最终用户文档提供了直接的更新。这些文档对于使新组员尽快参加工作也很有价值。

单个文档比由某些分析、设计和实现方法而产生的所有文档更合理。虽然一个较小的文档缺乏印象，但它是“活的”。相反，例如一个分析文档只是对于项目的特殊阶段有价值，然后很快变得陈旧了。对于知道将被丢掉的文档，人们很难对它再作更大的努力。

1.5 其他方法

当前有大量的形式化方法（不下20种）可用，由程序员选择[1]。有一些并不完全独立，它们有共同的思想基础，在某个更高层上，它们都是相同的。在最低层，很多方法都受语言的缺省表现约束，所以每个方法大概只能满足简单的项目。真正的好处是在较高层上，一个方法可用于实时硬件控制器，但可能不容易适合档案数据库的设计。

每种方法都有它的啦啦队，在我们对大规模方法采用之前应当更好地懂得它的语言基础，

[1] 这些是下书中的综述：Object Analysis and Design:Description of Methods, edited by Andrew T.F.Hutt of Object Management Group(OMG),john Wiley & Sons, 1994。

以此来认识一个方法是否适合我们的特殊风格，或者我们是否真需要一个方法。下面是对最流行的三种方法的描述，主要是供参考，不是购买比较。如果读者想了解更多的方法，有许多书籍可以参考，还有许多学习班可以参加。

1.5.1 Booch

Booch方法 [1] 是最早、最基本和最广泛被引用的一个方法。因为它是较早发展的，所以对各种程序设计问题都有意义。它的焦点是在OOP的类、方法和继承的单独性能上，步骤如下：

1. 在抽象的特定层上确定类和对象

这是可预见的一小步。我们用自然语言声明问题和解，并且确定关键特性，例如形成类的基本名词。如果我们在烟花行业，可能想确定工人、鞭炮、顾客，进而，可能需要确定化学药剂师、组装者、处理者，业余鞭炮爱好者和专业鞭炮者、购买者和观众。甚至更专门的，能确定年轻观众、老年观众、少年观众和父母观众。

2. 确定它们的语义

这是在相应的抽象层上定义类。如果我们计划创建一个类，我们应当确定类的相应观众。例如，如果创建鞭炮类，那么谁将观察它，化学药剂师还是观众？前者想知道在结构中有什么化学药剂，而后者将对鞭炮爆炸时释放的颜色和形状感兴趣。如果化学药剂师问起一个鞭炮产生主颜色的化学药剂，最好不要回答“有点冷绿和红”。同样，观众会对鞭炮点火后喷出的只是化学方程式感到迷惑不解。也许，我们的程序是为了眼前的市场，化学药剂师和观众都会用它，在这种情况下，鞭炮应当有主题属性和客体属性，并且能以和观察者相应的外观出现。

3. 确定它们之间的关系（CRC 卡片）

定义一个类如何与其他类互相作用。将关于每个类的信息制成表格的常见的方法是使用类，责任，协作（Class, Responsibility, Collaboration, CRC)卡片。这是一种小卡片（通常是一个索引卡），在它上面写上这个类的状态变量、它的责任（也就是它发送和接受的消息）和对与它互相作用的其他类的引用。为什么需要索引卡片？理由是我们应当能在一张小卡片上存放我们需要知道的关于一个类的所有内容，如果不能满足这点，那么这个类就太复杂了。理想的类应当能在扫视间被理解，索引卡片不仅实际可行，而且刚好符合大多数人思考问题合理的信息量。一种不涉及主要技术改革的解决办法对于每个人都可用的（就像本章前面描述的草稿方法中的文档结构化一样)。

4. 实现类

现在我们知道做什么了，投入精力进行编码。在大多数项目中， 编码将影响设计。

5. 反复设计

这样的设计过程给人一种类似著名的程序开发瀑布流方法的感觉。现在对这种方法的看法是有分歧的。在第一遍查看主要的抽象是否可以将类清晰地分离之后，可能需要重复前三个步骤。Booch写了一个“粗糙旅程完型设计过程”。如果这些类真正反映了这个解的自然语言描述，那么程序有完型的观点应当是可能的。也许要记住的最重要的事情是，通过缺省 —— 实际上是通过定义，如果修改了一个类，它的超类和子类应当仍然工作。不需要害怕修改，它不会破坏程序，对结果的任何改变将限制在子类和 /或被改变的这个类的特定协作者中。为了这个类而对CRC卡片的扫视也许是我们需要检验新版本的唯一线索。

---

[1] 参看Object-Oriented Design with Applications by Grady Booch, Benjamin Cummings, 1991.有关C++更新的版本。

1.5.2 责任驱动的设计（RDD）

这个方法[1]也用CRC卡片。在这里，正如名称蕴涵，卡片的重点在于责任的授权，而不是外观。这可由下面例子说明：Booch方法可以产生雇员——银行雇员——银行经理的继承，而在RDD中，这可能出现经理——金融经理——银行经理的继承。银行经理的主要责任是经理的责任，所以这种继承反映了这种关系。

更形式化地说，RDD包含如下内容：

1) 数据或状态： 对每个类的数据或状态变量的描述。

2) 池和源： 数据池和源的标识；处理或产生数据的类。

3) 观察者或观点： 观点或观察者类，用以隔离硬件依赖。

4) 辅助工具或帮助器： 辅助工具或帮助器类，例如连接表，它们包含很少的状态信息并简单地帮助其他类工作。

1.5.3 对象建模技术（OMT）

对象建模技术[2]（OMT）对过程增加一个或多个复杂层。 Booch方法强调类的功能表现，简单地定义它们作为自然语言解的轮廓。 RDD进了一步，强调类的责任超过强调类的表现。OMT用详细地绘制图表的方法，不仅描述类，而且还描述系统的各种状态，如下所述：

1) 对象模型，“什么”，对象图表： 该对象模型类似于由Booch方法和RDD产生的那些模型；对象的类通过责任相关联。

2) 动态模型，“何时”，状态图表： 该动态模型描写了系统的与时间有关的状态。不同的状态是通过转变相关联的；包含时间有关状态的一个例子是实时传感器，它从外部世界收集数据。

3) 功能模型，“如何”，数据流程表： 该功能模型跟踪数据流。它的理论是，在程序的最低层上，实际的工作是通过使用过程而完成的，因此程序的低层行为最好通过画数据流来理解，而不是通过画它的对象来理解。

1.6 为向OOP转变而采取的策略

如果我们决定采用OOP，我们的下一个问题可能是“我如何才能使得管理员、同事、部门、伙伴开始使用OOP？”想一想，作为独立的程序员，应当如何学习使用新语言和新程序设计。正如我们以前所做的，首先训练和做例子，再通过一个试验项目得到一个基本的感觉，不要做太混乱的事情，然后尝试做一个“真实世界”的实际有用的项目。在我们的第一个项目中，我们通过读、向上司问问题、与朋友切磋等方式，继续我们的训练。基本上，这就是许多作者建议的从C转到C++的方法。转变整个公司当然应当采用某个动态的小组，但回忆个人是如何做这件事的，能在转变的每一步中起到有益的作用。

1.6.1 逐步进入OOP

当向OOP和C++转变时，有一些方针要考虑：

1. 训练

第一步是一些形式的培训。记住公司在平常的C代码上的投资，并且当每个人都在为这些遗留的东西而为难时，应努力在6到9个月内不使公司完全陷入混乱。对一个小组进行培训，更

[1] 参看Designing Object-Oriented Software by Rebecca Wirfs-Brock et al., Prentice Hall, 1990。

[2] 参看Object-Oriented Modeling and Design by James Rumbaugh et al., Prentice Hall,1991。

适宜的情况是，这个小组成员是一些勤奋好学、能很好地在一起工作的人们，当他们学习 C++时，能形成他们自己的支持网。

有时建议采用另一种方法，即在整个公司层一起训练，包括为策略管理员而开设的概论课程，以及为项目建设者而开设的设计课程和编程课程。对于较小的公司或较大公司的部门，用他们做事情的方法去做基本的改变是非常好的。然而代价较高，所以一些公司可能选择以项目层训练开始，做飞行员式的项目（可能请一个外面的导师），然后让这个项目组变成公司其他人的老师。

2. 低风险项目

首先尝试一个低风险项目，并允许出错。一旦我们已经得到了一些经验，我们就将这第一个小组的成员安插到其他项目组中，或者用这个组的成员作为 OOP的技术支柱。第一个项目可能不能正确工作，所以该项目在事情的安排上应当不是非常重要的。它应当是简单的、自包含的和有教学意义的。这意味着它应当包括创建对公司的其他程序员学习 C++有意义的类。

3. 来自成功的模型

从草稿开始之前，挑一些好的面向对象设计的例子。很可能有些人已经解决过我们的问题，如果他们还没有正确地解决它，我们可以应用已经学到的关于封装的知识，来修改存在的设计，以适合我们自己的需要。这是设计模式 [1] 的一般概念。

4. 使用已存在的类库

转变为C++的主要经济动机是容易使用以类库形式存在的代码，最短的应用开发周期是除了main()以外不必自己写任何东西。然而，一些新程序员不理解这个，不知道已存在的类库，或由于对语言的迷恋希望写可能已经存在的类。如果我们努力查找和重用其他人在转变过程中的早期代码，那么我们在OOP和C++方面将是最优的。

5. 不要用C++重写已存在的代码

虽然用C++编译C代码通常会有（有时是很大的）好处，它能发现老代码中的问题，但是把时间花在对已存在的功能代码进行 C++重写上，通常不是时间的最佳利用。如果代码是为重用而编写的，会有很大的好处。但是，有可能出现这种情况：在最初的几个项目中，并不能看到效率如您梦想的一样增长，除非这是新项目。当然，如果项目是从头开始的， C++和OOP最好。

1.6.2 管理障碍

对于管理员，他的任务就是为他的小组获得资源，使他的小组克服通往成功路上的障碍。并且，他应当努力创造高产的和令人愉快的环境，以使得他的小组最大可能完成他所要求的奇迹。转变到C++的过程包含有下面的三类障碍，如果不花费何代价，那真是奇怪的。虽然对于C程序员（也许对于其他过程语言的程序员）小组，可证明选择 C++比选择其他OOP语言代价低，但这也不是免费的。我们应当认识到，在我们试图动员我们的公司转移到 C++和着手转移之前，还有一些障碍。

1. 启动代价

这个代价要比得到C++编译器大得多。如果投资培训（也许为了指导第一个项目），并且如果选购解决问题的类库，而不是试图自己建立这些类库，那么可以将中长期代价减到最低。这些都是很花钱的，必须在制定计划时充分考虑。另外，当学习新语言连同程序设计环境时，

[1] 参看Camma et al., 出处同上。

还会有损失效率的隐含代价。培训和指导确实能使这些代价降到最低，但是组员们必须克服他们自己的困惑去理解这些问题。在这个过程中，他们将会犯更多的错误（这是一个特征，因为失败是成功之母）和有更低的效率。尽管如此，对于一些类型的程序设计问题、正确的类和正确的开发环境，即使我们还在学习C++，也可能比我们仍然用C语言时有更高的效率（即便考虑到我们正在犯更多的错误和每天写更少行的代码）。

2. 性能问题

一个共同的问题是“ OOP不会使我的程序更大且更慢吗？ ”回答是“ 不一定 ”。大多数传统的OOP语言是在实验和在头脑中快速建立原型的情况下设计的，而不侧重于普通操作。这样，它们就本质地决定了在规模上的明显增长和在速度上的明显降低。然而， C++是在已有生产程序的情况下设计的。当我们的焦点是建立快速原型时，我们可以尽快地把构件拉在一起，而忽略效率问题。如果我们正在使用任何第三方库，这些库通常已经被厂商优化过了，我们用快速开发方法时，无论如何这都不成问题。我们有一个我们喜欢的系统时，如果它足够小和足够快，这就行了。如果达不到这种要求，我们就开始用一个有益的工具进行调整，首先寻求加速，这可以通过简单地运用建立在C++中的一些特性做到。如果这还不行，应当寻求修改低层来实现，所以使用特定类的原有代码不需要改变。只有这一切都无法解决这个问题时才需要改变设计。性能在设计中的地位如此重要，以致于这一因素必然是主要设计标准的指标之一。通过快速原型较早地发现这一点是有好处的。

如本章较早所述， C和C++之间在规模和速度上的差距常常是± 10%，并且通常更接近。实际上我们可以用C++得到在规模和速度上超过C的系统，因为我们为C++ 所做的设计可以完全不同于为C所做的设计。

在C和C++之间比较规模和速度的证据至今还只是估计，也许还会继续如此。尽管一些人建议对相同的项目用C和C++同时做，但也许不会有公司把钱浪费在这里，除非它非常大并且对这个研究项目感兴趣。即便如此，它也希望钱花得更好。已经从C（或其他过程语言）转到C++的程序员几乎一致都有在程序设计效率上很大提高的个人经验，这是我们能找到的最引人注目的证据。

3. 普遍的设计错误

当项目组开始使用OOP和C++时，程序员们将会出现一系列普遍的设计错误。这经常会发生，因为在早期项目的设计和实现过程中从专家们那里得到的反馈太少，公司中没有专家。程序员似乎会觉得，在这个周期中，他懂得OOP太早了并开始了一条不好的道路。有时，对这个语言有经验的人认为显而易见的事情可能是新手们在内部激烈争论的主题。大量的这类问题都能通过聘用外部专家培训和指导来避免。

1.7 小结

这一章希望读者对面向对象程序设计和 C++的广泛问题有一定的感性认识，包括为什么OOP是不同的；为什么C++特别不同；什么是OOP方法的概念和为什么应当（或不应当）使用一个概念；提出最小化方法的建议（这是我发展的方法），它允许以最小费用开始OOP项目；对其他方法的讨论；最后当公司转到OOP和C++时会遇到的各种问题。

OOP和C++可能不会对每个人都适合。对自己的需要做出估计，并决定 C++是否能很好地满足自己的要求，或者是否能很好地离开别的程序设计系统，这是很重要的。如果程序员知道，他的需要对可以预见的未来是非常特别的，如果他有特殊的约束，不能由 C++满足，那么可以用它研究替代物。即便他最终选择了C++作为他的语言，他至少应当懂得这些选项是什么，并应当对为什么取这个方向有清晰的看法。

第2章　数 据 抽 象

C++是一个能提高效率的工具。为什么我们还要努力（这是努力，不管我们试图做的转变多么容易）使我们从已经熟悉且效率高的语言（在这里是 C语言）转到另一种新的语言上？而且使用这种新语言，我们会在确实掌握它之前的一段时间内降低效率。这归因于我们确信通过使用新工具将会得到更大的好处。

用程序设计术语，多产意味着用较少的人在较少的时间内完成更复杂和更重要的程序。然而，选择语言时确实还有其他问题，例如运行效率（该语言的性质引起代码臃肿吗？）、安全性（该语言能有助于我们的程序做我们计划的事情并具有很强的纠错能力吗？）、可维护性（该语言能帮助我们创建易理解、易修改和易扩展的代码吗？）。这些都是本书要考察的重要因素。

简单地讲，提高生产效率，意味着本应花费三个人一星期的程序，现在只需要花费一个人一两天的时间。这会涉及到经济学的多层次问题。生产效率提高了，我们很高兴，因为我们正在建造的东西功能将会更强；我们的客户（或老板）很高兴，因为产品生产又快，用人又少；我们的顾客很高兴，因为他们得到的产品更便宜。而极大提高效率的唯一办法是使用其他人的代码，即使用库。

库，简单地说就是一些人已经写的代码，按某种方式包装在一起。通常，最小的包是带有扩展名如LIB的文件和向编译器声明库中有什么的一个或多个头文件。连接器知道如何在 LIB文件中搜索和提取相应的已编译的代码。但是，这只是提供库的一种方法。在跨越多种体系结构的平台上，例如UNIX，通常，提供库的最明智的方法是用源代码，这样在新的目标机上它能被重新编译。而在微软Windows上，动态连接库是最明智的方法，这使得我们能够利用新发布的DDL经常修改我们的程序，我们的库函数销售商可能已经将新 DDL发送给我们了。

所以，库大概是改进效率的最重要的方法。 C++的主要设计目标之一是使库容易使用。这意味着，在C中使用库有困难。懂得这一点就对 C++设计有了初步的了解，从而对如何使用它有了更深入的认识。

2.1　声明与定义

首先，必须知道“ 声明 ”和“ 定义 ”之间的区别，因为这两个术语在全书中会被确切地使用。“ 声明 ”向计算机介绍名字，它说，“ 这个名字是什么意思 ”。而“ 定义 ”为这个名字分配存储空间。无论涉及到变量时还是函数时含义都一样。无论在哪种情况下，编译器都在“ 定义 ”处分配存储空间。对于变量，编译器确定这个变量占多少存储单元，并在内存中产生存放它们的空间。对于函数，编译器产生代码，并为之分配存储空间。函数的存储空间中有一个由使用不带参数表或带地址操作符的函数名产生的指针。

定义也可以是声明。如果该编译器还没有看到过名字 A，程序员定义int A，则编译器马上为这个名字分配存储地址。

声明常常使用于extern关键字。如果我们只是声明变量而不是定义它，则要求使用 extern。对于函数声明， extern是可选的，不带函数体的函数名连同参数表或返回值，自动地作为一个声明。

函数原型包括关于参数类型和返回值的全部信息。 int f(float,char);是一个函数原型，因为它不仅介绍f这个函数的名字，而且告诉编译器这个函数有什么样的参数和返回值，使得编译器能对参数和返回值做适当的处理。 C++要求必须写出函数原型，因为它增加了一个重要的安全层。下面是一些声明的例子。

```
/*: DECLARE.C -- Declaration/definition examples */
extern int i; /* Declaration without definition */
extern float f(float); /* Function declaration */

float b;  /* Declaration & definition */
float f(float a) {  /* Definition */
  return a + 1.0;
}

int i; /* Definition */
int h(int x) { /* Declaration & definition */
  return x + 1;
}

main() {
  b = 1.0;
  i = 2;
  f(b);
  h(i);
}
```

在函数声明时，参数名可给出也可不给出。而在定义时，它们是必需的。这在 C语言中确实如此，但在C++中并不一定。

全书中，我们会注意到，每个文件的第一行是一个注释，它以注释符开始，后面跟冒号。这是我用的技术，可以利用诸如“ grep ”和“ awk ”这样的文本处理工具从代码文件中提取信息。在第一行中还包含有文件名，因此能在文本和其他文件中查阅这个文件，本书的代码磁盘中也很容易定义这个文件。

2.2 一个袖珍C库

一个小型库通常以一组函数开始，但是，已经用过别的 C 库的程序员知道，这里通常有更多的东西，有比行为、动作和函数更多的东西。还有一些特性（颜色、重量、纹理、亮度），它们都由数据表示。在C语言中，当我们处理一组特性时，可以方便地把它们放在一起，形成一个struct。特别是，如果我们想表示我们的问题空间中的多个类似的事情，则可以对每件事情创建这个struct的一个变量。

这样，在大多数C库中都有一组struct和一组活动在这些struct上的函数。现在看一个这样的例子。假设有一个程序设计工具，当创建时它的表现像一个数组，但它的长度能在运行时建立。我称它为stash。

```
/*: LIB.H -- Header file: example C library */
/* Array-like entity created at run-time */

typedef struct STASHtag {
  int size;  /* Size of each space */
  int quantity; /* Number of storage spaces */
  int next; /* Next empty space */
  /* Dynamically allocated array of bytes: */
  unsigned char* storage;
} Stash;

void initialize(Stash* S, int Size);
void cleanup(Stash* S);
int add(Stash* S, void* element);
void* fetch(Stash* S, int index);
int count(Stash* S);
void inflate(Stash* S, int increase);
```

在结构内部需要引用这个结构时可以使用这个 struct的别名，例如，创建一个链表，需要指向下一个struct的指针。在C库中，几乎可以在整个库的每个结构上看到如上所示的 typedef。这样做使得我们能把struct作为一个新类型处理，并且可以定义这个struct的变量，例如：

```
stash A, B, C;
```

注意，这些函数声明用标准 C 风格的函数原型，标准 C 风格比“老”C 风格更安全和更清楚。我们不仅介绍了函数名，而且还告诉编译器参数表和返回值的形式。

storage指针是一个unsigned char*。这是 C 编译器支持的最小的存储片，尽管在某些机器上它可能与最大的一般大，这依赖于具体实现。人们可能认为，因为 stash被设计用于存放任何类型的变量，所以void*在这里应当更合适。然而，我们的目的并不是把它当作某个未知类型的块处理，而是作为连续的字节块。

这个执行文件的源代码（如果我们买了一个商品化的库，我们可能得到的只是编译好的OBJ或LIB或DDL等）如下：

```
/*: LIB.C -- Implementation of
    example C library */
/* Declare structure and functions: */
#include "..\1\lib.h"
/* Error testing macros: */
#include <assert.h>
/* Dynamic memory allocation functions: */
#include <stdlib.h>
#include <string.h> /* memcpy() */
#include <stdio.h>
void initialize(Stash* S, int Size) {
  S->size = Size;
  S->quantity = 0;
  S->storage = 0;
  S->next = 0;
```

```
}

void cleanup(Stash* S) {
  if(S->storage) {
       puts("freeing storage");
       free(S->storage);
  }
}

int add(Stash* S, void* element) {
  /* enough space left? */
  if(S->next >= S->quantity)
    inflate(S, 100);
  /* Copy element into storage,
  starting at next empty space: */
  memcpy(&(S->storage[S->next * S->size]),
         element, S->size);
  S->next++;
  return(S->next - 1); /* Index number */
}

void* fetch(Stash* S, int index) {
  if(index >= S->next || index < 0)
    return 0;  /* Not out of bounds? */
  /* Produce pointer to desired element: */
  return &(S->storage[index * S->size]);
}

int count(Stash* S) {
  /* Number of elements in stash */
  return S->next;
}

void inflate(Stash* S, int increase) {
  void* v =
    realloc(S->storage,
            (S->quantity + increase)
            * S->size);
  /* Was it successful? */
  assert(v);
  S->storage = v;
  S->quantity += increase;
}
```

注意本地的 #include 风格，尽管这个头文件在本地目录下，但仍然以相对于本书的根目录给出。这样做，可以创建不同于这本书根目录的另外的目录，很容易拷贝文件到这个新目录下去实验，而不必担心改变 #include 中的路径。

initialize( )完成对 struct stash 的必要的设置，即设置内部变量为适当的值。最初，设置storage指针为零，设置size 指示器也为零，表示初始存储未被分配。

add( )函数在stash的下一个可用位子上插入一个元素。首先，它检查是否有可用空间，如果没有，它就用后面介绍的 inflate( ) 函数扩展存储空间。

因为编译器并不知道被存放的特定变量的类型（函数返回的都是 void*），所以我们不能只做赋值，虽然这的确是很方便的事情。代之，我们必须用标准 C 库函数memcpy( )一个字节一个字节地拷贝这个变量，第一个参数是 memcpy( ) 开始拷贝字节的目的地址，由下面表达式产生：

```
&(S->storage[S->next * S->size])
```

它指示从存储块开始的第next个可用单元结束。这个数实际上就是已经用过的单元号加一的计数，它必须乘上每个单元拥有的字节数，产生按字节计算的偏移量。这不产生地址，而是产生处于这个地址的字节，为了产生地址，必须使用地址操作符 &。

memcpy( )的第二和第三个参数分别是被拷贝变量的开始地址和要拷贝的字节数。 next计数器加一，并返回被存值的索引。这样，程序员可以在后面调用 fetch( )时用它来取得这个元素。

fetch( )首先看索引是否越界，如果没有越界，返回所希望的变量地址，地址的计算采用与add( )中相同的方法。

对于有经验的C程序员count( )乍看上去可能有点奇怪，它好像是自找麻烦，做手工很容易做的事情。例如，如果我们有一个 struct stash，例假设称为intStash，那么通过用intStash.next找出它已经有多少个元素的方法似乎更直接，而不是去做 count(&intStash)函数调用（它有更多的花费）。但是，如果我们想改变 stash的内部表示和计数计算方法，那么这个函数调用接口就允许必要的灵活性。并且，很多程序员不会为找出库的“ 更好 ”的设计而操心。如果他们能着眼于struct和直接取next的值，那么可能不经允许就改变 next。是不是能有一些方法使得库设计者能更好地控制像这样的问题呢？（是的，这是可预见的）。

动态存储分配

我们不可能预先知道一个stash需要的最大存储量是多少，所以由 storage指向的内存从堆中分配。堆是很大的内存块，用以在运行时分一些小单元。在我们写程序时，如果我们还不知道所需内存的大小，就可以使用堆。这样，我们可以直到运行时才知道需要存放 200个airplane变量，而不仅是 20个。动态内存分配函数是标准 C 库的一部分，包括 malloc( )、 calloc( )、realloc( )和free( )。

inflate( )函数使用realloc( )为stash得到更大的空间块。realloc( )把已经分配而又希望重分配的存储单元首地址作为它的第一个参数（如果这个参数为零，例如 initialize( )刚刚被调用时，realloc( )分配一个新块）。第二个参数是这个块新的长度，如果这个长度比原来的小，这个块将不需要作拷贝，简单地告诉堆管理器剩下的空间是空闲的。如果这个长度比原来的大，在堆中没有足够的相临空间，所以要分配新块，并且要拷贝内存。 assert( )检查以确信这个操作成功。（如果这个堆用光了， malloc( )、calloc( )和realloc( )都返回零。）

注意，C 堆管理器相当重要，它给出内存块，对它们使用 free( )时就回收它们。没有对堆进行合并的工具，如果能合并就可以提供更大的空闲块。如果程序多次分配和释放堆存储，最终会导致这个堆有大量的空闲块，但没有足够大且连续的空间能满足我们对内存分配的需要。但是，如果用堆合并器移动内存块，又会使得指针保存的不是相应的值。一些操作环境，例如Microsoft Windows有内置的合并，但它们要求我们使用专门的内存句柄（它们能临时地翻转为

指针，锁住内存后，堆压紧器不能移动它），而不是使用指针。

assert( )是在ASSERT.H中的预处理宏。assert( )取单个参数，它可以是能求得真或假值的任何表达式。这个宏表示：“我断言这是真的，如果不是，这个程序将打印出错信息，然后退出。”不再调试时，我们可以用一个标志使得这个断言被忽略。在调试期间，这是非常清楚和简便的测试错误的方法。不过，在出错处理时，它有点生硬：“对不起，请进行控制。我们的 C 程序对一个断言失败，并且跳出去。”在第17章中，我们将会看到， C++是如何用出错处理来处理重要错误的。

编译时，如果在栈上创建一个变量，那么这个变量的存储单元由编译器自动开辟和释放。编译器准确地知道需要多少存储容量，根据这个变量的活动范围知道这个变量的生命期。而对动态内存分配，编译器不知道需要多少存储单元，不知道它们的生命期，不能自动清除。因此，程序员应负责用 free( )释放这块存储， free( )告诉堆管理器，这个存储可以被下一次调用的malloc( )、calloc( )或realloc( )重用。合理的方法是使用库中cleanup( )函数，因为在这里，该函数做所有类似的事情。

为了测试这个库，让我们创建两个stash。第一个存放int，第二个存放80个字符的数组（我们可以把它看作新数据类型）。

```
/*: LIBTESTC.C -- Test demonstration library */
#include "..\1\lib.h"
#include <stdio.h>
#include <assert.h>
#define BUFSIZE 80

main() {
  Stash intStash, stringStash;
  int i;
  FILE* file;
  char buf[BUFSIZE];
  char* cp;
/* .... */
initialize(&intStash, sizeof(int));
for(i = 0; i < 100; i++)
  add(&intStash, &i);
/* Holds 80-character strings: */
initialize(&stringStash,
           sizeof(char) * BUFSIZE);
file = fopen("LIBTESTC.C", "r");
assert(file);
while(fgets(buf, BUFSIZE, file))
  add(&stringStash, buf);
fclose(file);

for(i = 0; i < count(&intStash); i++)
  printf("fetch(&intStash, %d) = %d\n", i,
         *(int*)fetch(&intStash, i));
```

```
  i = 0;
  while((cp = fetch(&stringStash, i++)) != 0)
    printf("fetch(&stringStash, %d) = %s",
           i - 1, cp);
  putchar('\n');
  cleanup(&intStash);
  cleanup(&stringStash);
}
```

在main( )的开头定义了一些变量，其中包括两个stash结构变量，当然。稍后我们必须在这个程序块的对它们初始化。库的问题之一是我们必须向用户认真地说明初始化和清除函数的重要性，如果这些函数未被调用，就会出现许多问题。遗憾的是，用户不总是记得初始化和清除是必须的。他们只知道他们想完成什么，并不关心我们反复说的："喂，等一等，您必须首先做这件事。"一些用户甚至认为初始化这些元素是自动完成的。的确没有机制能防止这种情况的发生（只有多预示）。

intStash适合于整型，stringStash适合于字符串。这些字符串是通过打开源代码文件LIBTEST.C 和把这些行读到 stringStash 而产生的。注意一些有趣的地方：标准 C 库函数打开和读文件所使用的技术与在stash中使用的技术类似。fopen( )返回一个指向 FILE struct的指针，这个 FILE struct是在堆上创建的，并且能将这个指针传给涉及到这个文件的任何函数。（在这里是fgets( )）。fclose( )所做的事情之一是向堆释放这个FILE struct。一旦我们开始注意到这种模式的，包含着struct和有关函数的 C 库后，我们就能到处看到它。

装载了这两个stash之后，可以打印出它们。intStach的打印用一个for循环，用count( )确定它的限度。stringStash的打印用一个while语句，如果fetch( )返回零则表示打印越界,这时跳出循环。

在我们考虑有关 C 库创建的问题之前，应当了解另外一些事情（我们可能已经知道这些，因为我们是 C 程序员）。第一，虽然这里用头文件，而且实际上用得很好，但它们不是必须的。在C中可能会调用还未声明的函数。好的编译器会告诫我们应当首先声明函数，但不强迫这样做。这是很危险的，因为编译器能假设以 int参数调用的函数有包含 int的参数表，并据此处理它，这是很难发现的错误。

注意，头文件 LIB.H 必须包含在涉及 stash 的所有文件中，因为编译器不可能猜出这个结构是什么样子的。它能猜出函数，即便它可能不应当这样，但这是 C 的一部分。

每个独立的C文件就是一个处理单元。就是说，编译器在每个处理单元上单独运行，而编译器在运行时只知道这个单元。这样，用包含头文件提供信息是相当重要的，因为它为编译器提供了对程序其他部分的理解。在头文件中的声明特别重要，因为无论是在哪里包含这个头文件，编译器都会知道要做什么。例如，若在一个头文件中声明void foo(float)，编译器就会知道，如果我们用整型参数调用它，它会自动把int转变为float。如果没有声明，这个编译器就会简单地猜测，有一个函数存在，而不会做这个转变。

对于每个处理单元，编译器创建一个目标文件，带有扩展名 .o 或 .obj 或类似的名字。必须再用连接器将这些目标文件连同必要的启动代码连接成可执行程序。在连接期间，所有的外部引用都必须确定。例如在 LIBTEST.C 中，声明并使用函数 initialize( ) 和fetch( )，(也就是，编译器被告知它们像什么，)但未定义。它们是在 LIB.C 中定义的,这样，在 LIBTEST.C 中的这些调用都是外部引用。当连接器将目标文件连接在一起时，它找出未确定的引用并寻找这些引

用对应的实际地址，用这些地址替换这些外部引用。

重要的是认识到，在 C 中，引用就是函数名，通常在它们前面加上下划线。所以，连接器所要做的就是让被调用的函数名与在目标文件中的函数体匹配起来。如果我们偶然做了一个调用，编译器解释为 foo(int)，而在其他目标文件中有 foo(float)的函数体，连接器将认为一个_foo在一处而另一个_foo在另一处，它会认为这都是对的。在调用 foo( )处将一个int放进栈中，而foo( )函数体期望在这个栈中的是一个 float。如果这个函数只读这个值而不对它写，尚不会破坏这个栈。但从这个栈中读出的 float值可能会有另外的某种理解。这是最坏的情况，因为很难发现这个错误。

2.3 放在一起：项目创建工具

分别编译时（把代码分成多个处理单元），我们需要一些方法去编译所有的代码，并告诉连接器把它们与相应的库和启动代码放在一起，形成一个可执行文件。大部分编译器允许用一条命令行语句。例如编译器命名为cpp，可写:

```
cpp libtest.c lib.c
```

这个方法带来的问题是，编译器必须首先编译每个处理单元，而不管这个单元是否需要重建。虽然我们只改变了一个文件，但却需要耗费时间来对项目中的每一个文件进行重新编译。

对这个问题的第一种解决办法，已由 UNIX（C的诞生地）提出，是一个被称为 make 的程序。make比较源代码文件的日期和目标文件的日期，如果目标文件的日期比源代码文件的早，make 就调用这个编译器对这个单元进行处理。我们可以从编译器文档 [1] 中学到更多的关于 make 的知识。

make 是有用的，但学习和配置 makefile 有点乏味。makefile 是描述项目中所有文件之间关系的文本文件。因此，编译器销售商发行它们自己的项目创建工具。这些工具向我们询问项目中有哪些处理单元，并确定它们的关系。这些关系有些类似于 makefile文件，通常称为项目文件。程序设计环境维护这个文件，所以我们不必为它担心。项目文件的配置和使用随系统而异，假设我们正在使用我们选择的项目创建工具来创建程序，我们会发现如何使用它们的相应文档（虽然由编译器销售商提供的项目文件工具通常是非常简单的，可以不费劲地学会它们）。

文件名

应当注意的另一个问题是文件命名。在 C中，惯例是以扩展名 .h命名头文件（包含声明），以 .c 命名实现文件（它引起内存分配和代码生成）。C++ 继续演化。它首先是在 Unix 上开发的，这个操作系统能识别文件名的大小写。原来的文件名简单地变为大写，形成 .H 和 .C 版本。这样对于不区分大小写的操作系统，例如 MS-DOS，就行不通了。DOS C++ 厂商对于头文件和实现文件分别使用扩展名.hxx 和 .cxx。后来，有人分析出，需要不同扩展名的唯一原因是使得编译器能确定编译C还是C++文件。因为编译器不直接编译头文件，所以只有实现文件的扩展名需要改变。现在人们已经习惯于在各种系统上对于实现文件都使用 .cpp而对于头文件使用 .h。

2.4 什么是非正常

我们通常有特别的适应能力，即使是对本不应该适应的事情。 stash 库的风格对于 C 程序员已经是常用的了，但是如果观察它一会儿，就会发现它是相当笨拙的。因为在使用它时，必

[1] 参看由作者编写的C++ Inside & Out，(Osborne/McGraw-Hill,1993)。

须向这个库中的每一个函数传递这个结构的地址。而当读这些代码时，这种库机制会和函数调用的含义相混淆，试图理解这些代码时也会引起混乱。

然而在 C 中，使用库的最大的障碍是名字冲突问题。C 对于函数使用单个名字空间，所以当连接器找一个函数名时，它在一个单独的主表中查找，而当编译器在单个处理单元上工作时，它只能对带有某些特定名字的函数进行处理工作。

现在假设要支持从不同的厂商购买的两个库，并且每个库都有一个必须被初始化和清除的结构。两个厂商都认为initialize( )和cleanup( )是好名字。如果在某个处理单元中同时包含了这两个库文件，C 编译器怎么办呢？幸好，标准 C 给出一个出错，告诉在声明函数的两个不同的参数表中类型不匹配。即便不把它们包含在同一个处理单元中，连接器也会有问题。好的连接器会发现这里有名字冲突，但有些编译器仅仅通过查找目标文件表，按照在连接表中给出的次序，取第一个找到的函数名（实际上，这可以看作是一种功能，因为可以用自己的版本替换一个库函数）。

无论哪种情况，都不允许使用包含具有同名函数的两个库。为了解决这个问题， C 库厂商常常会在它们的所有函数名前加上一个独特字符串。所以， initialize( )和cleanup( )可能变为stash_initialize( )和stash_cleanup( )。这是合乎逻辑的，因为它“分解了”这个struct 的名字，而该函数以这样的函数名对这个struct操作。

现在，迈向 C++第一步的时候到了。我们知道， struct 内部的变量名不会与全局变量名冲突。而当一些函数在特定 struct上运算时，为什么不把这一优点扩展到这些函数名上呢？也就是，为什么不让函数是 struct 的成员呢？

2.5 基本对象

C++的第一步正是这样。函数可以放在结构内部，作为“成员函数”。在这里 stash 是：

```
//: LIBCPP.H -- C library converted to C++

struct stash {
  int size;  // Size of each space
  int quantity; // Number of storage spaces
  int next; // Next empty space
   // Dynamically allocated array of bytes:
  unsigned char* storage;
  // Functions!
  void initialize(int Size);
  void cleanup();
  int add(void* element);
  void* fetch(int index);
  int count();
  void inflate(int increase);
};
```

首先注意到的可能是新的注释文法//。这是对 C 风格注释的补充， C 原来的注释文法仍然能用。C++注释直到该行的结尾，它有时非常方便。另外，我们会在这本书中，在文件的第一行的//之后加一个冒号，后面跟的是这个文件名和简要描述。这就可以了解代码所在的文件。并且还可以很容易地用本书列出的名字从电子源代码中识别出这个文件。

其次，注意到这里没有typedef。在C++中，编译器不要求我们创建typedef，而是直接把结构名转变为这个程序的新类型名（就像int、char、float、double一样）。stash 的用法仍然相同。

所有的数据成员与以前完全相同，但现在这些函数在 struct的内部了。另外，注意到，对应于这个库的 C 版本中第一个参数已经去掉了。在C++中，不是硬性传递这个结构的地址作为在这个结构上运算的所有函数的一个参数，而是编译器背地里做了这件事。现在，这些函数仅有的参数与这些函数所做的事情有关，而不与这些函数运算的机制有关。

认识到这些函数代码与在 C 库中的那些同样有效，是很重要的。参数的个数是相同的（即便我们还没有看到这个结构地址被传进来，实际上它在这里），并且每个函数只有一个函数体。正因为如此，写：

```
stash A, B, C;
```

并不意味着每个变量得到不同的add( )函数。

被产生的代码几乎和我们已经为 C 库写的一样。有趣的是，这同时就包括了为过程stash_initialize( )、stash_cleanup( )等所做的“名字分解”。当函数在 struct 内时，编译器有效地做了相同的事情。因此，在 stash 内部的 initialize( ) 将不会与任何其他结构中的 initialize( ) 相抵触。大部分时间都不必为函数名字分解而担心——即使使用未分解的函数名。但有时还必须能够指出这个initialize( )属于这个 struct stash 而不属于任何其他的struct。特别是，定义这个函数时，需要完全指定它是哪一个。为了完成这个指定任务， C++有一个新的运算符::，即范围分解运算符（这样命名是因为名字现在能在不同的范围：在全局范围或在这个 struct 的范围）。例如，如果希望指定initialize( )属于 stash，就写stash::initialize(int Size, int Quantity)。对于stash 的C++版本，我们可以看到，在函数定义中如何使用范围分解运算符：

```
//: LIBCPP.CPP -- C library converted to C++
// Declare structure and functions:
#include "..\1\libcpp.h"
#include <assert.h> // Error testing macros
#include <stdlib.h> // Dynamic memory
#include <string.h> // memcpy()
#include <stdio.h>

void stash::initialize(int Size) {
  size = Size;
  quantity = 0;
  storage = 0;
  next = 0;
}

void stash::cleanup() {
  if(storage) {
    puts("freeing storage");
    free(storage);
  }
}

int stash::add(void* element) {
```

```
  if(next >= quantity) // Enough space left?
    inflate(100);
  // Copy element into storage,
  // starting at next empty space:
  memcpy(&(storage[next * size]),
         element, size);
  next++;
  return(next - 1); // Index number
}

void* stash::fetch(int index) {
  if(index >= next || index < 0)
    return 0;  // Not out of bounds?
  // Produce pointer to desired element:
  return &(storage[index * size]);
}

int stash::count() {
  return next; // Number of elements in stash
}

void stash::inflate(int increase) {
  void* v =
    realloc(storage, (quantity+increase)*size);
  assert(v);  // Was it successful?
  storage = (unsigned char*)v;
  quantity += increase;
}
```

这个文件有几个其他的事项要注意。首先，在头文件中的声明是由编译器要求的。在 C++ 中，不能调用未事先声明的函数，否则编译器将报告一个出错信息。这是确保这些函数调用在被调用点和被定义点之间一致的重要方法。通过强迫在调用之前必须声明， C++ 编译器可以保证我们用包含这个头文件的方式完成这个声明。如果我们在这个函数被定义的地方还包含有同样的头文件，则编译器将作一些检查以保证在这个头文件中的声明和这个定义匹配。这意味着，这个头文件变成了函数声明的有效的仓库，并且保证这些函数在项目中的所有处理单元中使用一致。

当然，全局函数仍然可以在定义和使用它的每个地方用手工方式声明（这是很乏味的，以致于变得不太可能）。然而，结构的声明必须在它们定义和使用之前，而放置结构定义的最习惯的位置是在头文件中，除非我们有意把它藏在代码文件中。

可以看到，除了范围分解和来自这个库的C版本的第一个参数不再是显式的这一事实以外，所有这些成员函数实际上都是一样的。当然，这个参数仍然存在，因为这个函数必须工作在一个特定的struct 变量上。但是，在成员函数内部，成员选择照常使用。这样，不写s->size = size，而写size = size。这就去除了并不能在此增加信息的冗余 s->。当然， C++ 编译器必须为我们做这些事情。实际上，它取“ 秘密 ”的第一个参数，并且当提到类的数据成员的任何时候，必须

应用成员选择器。这意味着，无论何时，当在另一个类的成员函数中时，我们可以通过简单地给出它的名字，提及任何成员（包括成员函数）。编译器在找出这个名字的全局版本之前先在局部结构的名字中搜索。这样这个性能意味着不仅代码更容易写，而且更容易阅读。

但是，如果因为某种原因，我们希望能够处理这个结构的地址，情况会怎么样呢？在这个库的 C 版本中，这是很容易的，因为每个函数的第一个参数是称作 S 的一个 stash* 。在 C++ 中，事情是更一致的。这里有一个特殊的关键字，称为 this，它产生这个struct的地址。它等价于这个库的 C 版本的 S 。所以我们可以用下面语句恢复成C风格。

```
this->size = Size;
```

对这种书写形式进行编译所产生的代码是完全一样的。通常，不经常用 this，只是需要时才使用。在这些定义中还有最后一个变化。在 C 库中的 inflate( )中，可以将void*赋给其他任何指针，例如

```
S->storage = v;
```

而且编译器能够通过。但在 C++ 中，这个语句是不允许的，为什么呢？因为在 C 中，可以给任何指针赋一个 void*（它是 malloc( )、calloc( )和 realloc( )的返回），而不需计算。C 对于类型信息不挑剔，所以它允许这种事情。 C++不同，类型在C++中是严格的，当类型信息有任何违例时，编译器就不允许。这一点一直是很重要的，而对于 C++尤其重要，因为在struct中有成员函数。如果我们能够在 C++ 中向struct传递指针而不被阻止，那么我们就能最终调用对于struct逻辑上并不存在的成员函数。这是防止灾难的一个实际的办法。因此， C++允许将任何类型的指针赋给 void*（这是 void* 的最初的意图，它要求 void* 足够大，以存放任何类型的指针），但不允许将void* 指针赋给任何其他类型的指针。 一项基本的要求是告诉读者和编译器，我们知道正在用的类型。这样，我们可以看到 calloc( ) 和 realloc( ) 的返回值严格地指派为(unsigned char*)。

这就带来了一个有趣的问题，C++的最重要的目的之一是能编译尽可能多的已存在的 C 代码，以便容易地向这个新语言过渡。那么在上述例子中如何使用标准 C 库函数？另外，所有的 C 运算符和表达式在 C++ 中都可用。但是，这并不意味着 C 允许的所有代码在 C++ 中也允许。有一些C编译器允许的东西是危险的和易出错的（本书中还会看到它们）。C++ 编译器对于这些情况产生警告和出错信息，这样将更有好处。实际上， C中有许多我们知道的错误只是不能找出它的原因，但是一旦用C++重编译这个程序，编译器就能指出这些问题。在 C 中，我们常常发现能使程序通过编译，然后我们必须再花力气使它工作。在 C++中，常常是，程序编译正确了，它也就能工作了。这是因为该语言对类型要求更严格的缘故。

在下面的测试程序中，可以看到 stash 的C++版本所使用的另一些东西。

```
//: LIBTEST.CPP -- Test of C++ library
#include "..\1\libcpp.h"
#include <stdio.h>
#include <assert.h>
#define BUFSIZE 80

main() {
  stash intStash, stringStash;
  int i;
  FILE* file;
  char buf[BUFSIZE];
```

```
  char* cp;
  // ....
  intStash.initialize(sizeof(int));
  for(i = 0; i < 100; i++)
    intStash.add(&i);
  // Holds 80-character strings:
  stringStash.initialize(sizeof(char)*BUFSIZE);
  file = fopen("LIBTEST.CPP", "r");
  assert(file);
  while(fgets(buf, BUFSIZE, file))
    stringStash.add(buf);
  fclose(file);

  for(i = 0; i < intStash.count(); i++)
    printf("intStash.fetch(%d) = %d\n", i,
           *(int*)intStash.fetch(i));

  i = 0;
  while(
    (cp = (char*)stringStash.fetch(i++))!=0)
      printf("stringStash.fetch(%d) = %s",
             i - 1, cp);
  putchar('\n');
  intStash.cleanup();
  stringStash.cleanup();
}
```

这些代码与原来的代码相当类似，但在调用成员函数时，在函数名字之前使用成员运算符“.”。这是一个传统的文法，它模仿了结构数据成员的使用。所不同的是这里是函数成员，有一个参数表。

当然，该编译器实际产生的调用，看上去更像原来的库函数。如果我们考虑名字分解和this传递，C++函数调用 intStash.initialize(sizeof(int), 100) 就和 stash_initialize(&intStash, sizeof(int), 100)一样了。如果我们想知道在内部所进行的工作，可以回忆最早的 C++编译器cfront，它由AT&T开发，它输出的是C代码，然后再由C 编译器编译。这个方法意味着 cfront能使C++很快地转到有 C 编译器的机器上，有助于传播C++ 编译器技术。

在下面语句中我们还会注意到类型转化的情况。

```
while(cp = (char*)stringStash.fetch(i++))
```

这是由于在 C++ 中有严格类型检查而导致的结果。

2.6 什么是对象

我们已经看到了一个最初的例子，现在回过头来看一些术语。把函数放进结构是 C++ 中的根本改变，并且这引起我们将结构作为新概念去思考。在 C 中，结构是数据的凝聚，它将数据捆绑在一起，使得我们可以将它们看作一个包。但这除了能使程序设计方便之外，别无其他好处。这些结构上的运算可以用在别处。然而将函数也放在这个包内，结构就变成了新的创

造物，它既能描述属性（就像 C中的 struct 能做的一样），又能描述行为，这就形成了对象的概念。对象是一个独立的有约束的实体，有自己的记忆和活动。

“对象”和“面向对象的程序设计”（OOP）术语不是新的。第一个OOP语言是 Simula-67，于1967年由 Scandinavia发明，用于辅助解决建模问题。这些问题似乎总是包括一束相同的实体（诸如人、细菌、小汽车），它们为互相交互而忙碌。 Simula允许对实体创建一般的描述，描写它的属性和行为，然后取总的一束。在 Simula 中，这种“一般的描述”称为 class （类）（一个将在后面章节中看到的术语）。由类产生的大量的项称为对象。在 C++ 中，对象只是一个变量，最纯的定义是“存储的一个区域”。它是能存放数据的空间，并隐含着还有在这些数据上的运算。

不幸的是，对于各种语言，涉及这些术语时，并不完全一致，尽管它们是可以接受的。我们有时还会遇到面向对象语言是什么的争论，虽然到目前为止这已被认为是相当好的选择。还有一些语言是object-based（基于对象的），意味着它们有像 C++ 的结构加函数这样的对象，正如我们已经看到的。然而，这只是到达面向对象语言历程中的一部分，停留在把函数捆绑在结构内部的语言是基于对象的，而不是面向对象的。

2.7 抽象数据类型

将数据连同函数捆绑在一起，这一点就允许创建新的类型。这常常被称为封装 [1]。一个已存在的数据类型，例如 float，有几个数据块，一个指数，一个尾数和一个符号位。我们能够告诉它：与另一个 float 或 int 相加，等等。它有属性和行为。

stash 也是一个新数据类型，可以add( )、fetch( )和 inflate( )。由说明stash S创建一个 stash 就像由说明 float f 创建一个 float 一样。一个 stash 也有属性和行为，甚至它的活动就像一个实数——一个内建的数据类型。我们称stash为抽象数据类型（abstract dada type），也许这是因为它能允许我们从问题空间把概念抽象到解空间。另外，C++编译器把它看作一个新的数据类型，并且如果说一个函数需要一个 stash，编译器就确保传递了一个 stash 给这个函数。对抽象数据类型（有时称为用户定义类型）的类型检查就像对内建类型的类型检查一样严格。

然而，我们会看到在对象上完成运算的方法有所不同。 object.member_function(arglist)是对一个对象“调用一个成员函数”。而在面向对象的用法中，也称之为“向一个对象发送消息”。这样，对于 stash S，语句 S.add(&i)“发送消息给 S”，也就是说，“对自己 add( )”。事实上，面向对象程序设计可以总结为一句话，“向对象发送消息”。需要做的所有事情就是创建一束对象并且给它们发送消息。当然，问题是勾画出我们的对象和消息是什么，但如果完成了这些，C++ 的实现就直截了当了。

2.8 对象细节

这时我们大概和大多数 C 程序员一样会感到有点困惑，因为原有的 C 是非常低层和面向效率的语言。在研讨会上经常提出的一个问题是“对象应当多大和它应当像什么”。回答是“最好莫过于和我们希望来自 C的 struct一样”。事实上，C struct （不带有 C++ 装饰）在由 C 和 C++ 编译器产生的代码上完全相同，这可以使那些关心代码的安排和大小细节的 C程序员放心了。并且，由于某种原因，直接访问结构的字节，而不是使用标识符，不一定是效率更高的办法。

[1] 应当知道，这个术语似乎是有争议的题目。一些人就像这里的用法一样用它，而另一些人用它描述隐藏的实现，这将在第三章中讨论。

一个结构的大小是它的所有成员大小的和。有时，当一个 struct 被编译器处理时，会增加额外的字节以使得捆绑更整齐，这主要是为了提高执行效率。在第 14章和第16章中，将会看到如何在结构中增加“秘密”指针，但是现在不必关心这些。

用sizeof运算可以确定 struct 的长度。这里有一个小例子：

```
//: SIZEOF.CPP -- Sizes of structs
#include <stdio.h>
#include "..\1\lib.h"
#include "..\1\libcpp.h"
struct A {
  int I[100];
};

struct B {
  void f();
};

void B::f() {}

main() {
  printf("sizeof struct A = %d bytes\n",
         sizeof(A));
  printf("sizeof struct B = %d bytes\n",
         sizeof(B));
  printf("sizeof Stash in C = %d bytes\n",
         sizeof(Stash));
  printf("sizeof stash in C++ = %d bytes\n",
         sizeof(stash));
}
```

第一个打印语句产生的结果是200，因为每个 int 占二个字节。struct B 是奇异的，因为它是没有数据成员的 struct。在 C 中，这是不合法的，但在 C++ 中，以这种选择方式创建一个 struct，唯一的目的就是划定函数名的范围，所以这是允许的。尽管如此，由第二个 printf ()语句产生的结果是一个有点奇怪的非零值。在该语言较早的版本中，这个长度是零，但是，当创建这样的对象时出现了笨拙的情况：它们与紧跟着它们创建的对象有相同的地址，没有区别。这样，无数据成员的结构总应当有最小的非零长度。

最后两个 sizeof 语句表明在 C++ 中的结构长度与 C 中等价版本的长度相同。C++ 尽力不增加任何花费。

2.9 头文件形式

当我第一次学习用 C 编程时，头文件对我是神秘的。许多有关 C语言的书似乎不强调它，并且编译器也并不强调函数声明，所以它在大部分时间内似乎是可要可不要的，除非要声明结构时。在 C++ 中，头文件的使用变得非常清楚。它们对于每个程序开发是强制的，在它们中放入非常特殊的信息：声明。头文件告诉编译器在我们的库中哪些是可用的。因为对于 CPP 文件能够不要源代码而使用库（只需要对象文件或库文件），所以头文件是存放接口规范的唯

一地方。

头文件是库的开发者与它的用户之间的合同。它说：“这里描述的是库能做什么。”它不说如何做，因为如何做存放在CPP文件中，开发者不需要分发这些描述“如何做”的源代码给用户。

该合同描述数据结构，并说明函数调用的参数和返回值。用户需要这些信息来开发应用程序，编译器需要它们来产生相应的代码。

编译器强迫执行这一合同，也就是要求所有的结构和函数在它们使用之前被声明，当它们是成员函数时，在它们被定义之前被声明。这样，就强制把声明放在头文件中并把这个头文件包含在定义成员函数的文件和使用它们的文件中。因为描述库的单个头文件被包括在整个系统中，所以编译器能保证一致和避免错误。

为了恰当地组织代码和写有效的头文件，有一些问题必须知道。第一个问题是将什么放进头文件中。基本规则是“只声明”，也就是说，对于编译器只需要一些信息以产生代码或创建变量分配内存。这是因为，在一个项目中，头文件也许会包含在几个处理单元中，而如果内存分配不止一个地方，则连接器会产生多重定义错误。

这个规则不是非常严格的。如果在头文件中定义“静态文件”的一段数据（只在文件内可视），在这个项目中将有这个数据的多个实例，编译器不会报错。基本上，不要在头文件中做在连接时会引起混淆的任何事情。

关于头文件的第二个问题是重复声明。C 和 C++ 都允许对函数重复声明，只要这些重复声明匹配，但决不允许对结构重复声明。在 C++ 中，这个规则特别重要，因为，如果编译器允许对结构重复声明而且这两个重复声明又不一样，那么应当使用哪一个呢？

重复声明问题在 C++ 中很少出现，因为每个数据类型（带有函数的结构）一般有自己的头文件。但我们如果希望创建使用某个数据类型的另一个数据类型，必须在另一个头文件中包含它的头文件。在整个项目中，很可能有几个文件包含同一个头文件。在编译期间，编译器会几次看到同一个头文件。除非做适当的处理，否则编译器将认为是结构重复声明。

典型的防止方法是使用预处理器隔离这个头文件。如果有一个头文件名为 FOO.H，一般用“名字分解”产生预处理名，以防止多次包含这个头文件。FOO.H 的内部可以如下：

```
#ifndef FOO_H_
#define FOO_H_
// Rest of header here ...
#endif // FOO_H_
```

注意：不用前导下划线，因为标准 C 用前导下划线指明保留标识符。

在项目中使用头文件

用 C++ 建立项目时，我们通常要汇集大量不同的类型（带有相关函数的数据结构）。一般将每个类型或一组相关类型放在一个单独的头文件中，然后在一个处理单元中定义这个类型的函数。当使用这个类型时必须包含这个头文件，形成适当的声明。

有时这个模式会在本书中使用，但如果例子很小，结构声明、函数定义和 main( ) 函数可以出现在同一个文件中。应当记住，在实际上使用的是隔离的文件和头文件。

2.10 嵌套结构

在全局名字空间之外为数据和函数取名字是有好处的，可以将这种方式推广到对结构的处理

中。我们可以将一个结构嵌套在另一个中，这就可以将相关联的元素放在一起。声明文法在下面结构中可以看到，这个结构用非常简单的链接表方式实现了一个栈，所以它决不会运行越界。

```
//: NESTED.H -- Nested struct in linked list
#ifndef NESTED_H_
#define NESTED_H_

struct stack {
  struct link {
    void* data;
    link* next;
    void initialize(void* Data, link* Next);
  } * head;
  void initialize();
  void push(void* Data);
  void* peek();
  void* pop();
  void cleanup();
};
#endif // NESTED_H_
```

这个嵌套 struct 称为 link，它包括指向这个表中的下一个 link 的指针和指向存放在 link 中的数据的指针，如果 next 指针是零，意味着表尾。

注意，head 指针紧接在 struct link 声明之后定义，而不是单独定义 link* head。这是来自C的文法，但它强调在结构声明之后的分号的重要性，分号表明这个结构类型的定义表结束（通常这个定义表是空的）。

正如到目前为止所有描述的结构一样，嵌套结构有它自己的 initialize( ) 函数。为了确保正确地初始化，stack 既有 initialize( ) 又有 cleanup( ) 函数。此外还有push( ) 函数，它取一个指向希望存放数据的存储单元（假设已经分配在堆中）；还有 pop( ),它返回栈顶的data指针，并去除栈顶元素（注意，我们对破坏 data 指针负有责任）；peek( )函数也从栈顶返回 data 指针，但是它在栈中保留这个栈顶元素。

cleanup 去除每个栈元素，并释放data 指针。

下面是一些函数的定义。

```
//: NESTED.CPP -- Linked list with nesting
#include <stdlib.h>
#include <assert.h>
#include "nested.h"

void stack::link::initialize(
    void* Data, link* Next) {
  data = Data;
  next = Next;
}

void stack::initialize() { head = 0; }
```

```
void stack::push(void* Data) {
  link* newlink = (link*)malloc(sizeof(link));
  assert(newlink);
  newlink->initialize(Data, head);
  head = newlink;
}

void* stack::peek() { return head->data; }

void* stack::pop() {
  if(head == 0) return 0;
  void* result = head->data;
  link* oldHead = head;
  head = head->next;
  free(oldHead);
  return result;
}

void stack::cleanup() {
  link* cursor = head;
  while(head) {
    cursor = cursor->next;
    free(head->data); // Assumes a malloc!
    free(head);
   head = cursor;
  }
}
```

第一个定义特别有趣，因为它表明如何定义嵌套结构的成员。简单地两次使用范围分解运算符，以指明这个嵌套 struct 的名字。stack::link::initialize( ) 函数取参数并把参数赋给它的成员们。虽然用手工做这些事情相当容易，但是，我们将来会看到这个函数的不同的形式，所以它更有意义。

stack::initialize( ) 函数置 head 为零，使得这个对象知道它有一个空表。

stack::push( ) 取参数，也就是一个指向希望用这个 stack 保存的一块数据的指针，并且把这个指针放在栈顶。首先，使用 malloc( ) 为 link 分配空间，link将插入栈顶。然后调用 initialize( ) 函数对这个 link 的成员赋适当的值。注意，next 指针赋为当前的 head，而 head 赋为新link 指针。这就有效地将 link 放在这个表的顶部了。

stack::pop( )取出当前在该栈顶部的 data 指针，然后向下移 head 指针，删除该栈老的栈顶元素。stack::cleanup( )创建cursor 在整个栈上移动，用free( )释放每个link的data和link 本身。

下面是测试这个栈的例子。

```
//: NESTEST.CPP -- Test of nested linked list
#include "..\1\nested.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
```

```
main(int argc, char** argv) {
  stack textlines;
  FILE* file;
  char* s;
  #define BUFSIZE 100
  char buf[BUFSIZE];
  assert(argc == 2); // File name is argument
  textlines.initialize();
  file = fopen(argv[1], "r");
  assert(file);
  // Read file and store lines in the stack:
  while(fgets(buf, BUFSIZE, file)) {
    char* string =(char*)malloc(strlen(buf)+1);
    assert(string);
    strcpy(string, buf);
    textlines.push(string);
  }
  // Pop the lines from the stack and print them:
  while((s = (char*)textlines.pop()) != 0) {
    printf("%s", s); free(s); }
  textlines.cleanup();
}
```

这个例子与前面一个非常类似，这些行存放到这个栈中，然后弹出它们，这会使这个文件被反序打印出。另外，要打开文件的文件名是从命令行中取出的。

全局范围分解

编译器通过缺省选择的名字（“最近”的名字）可能不是我们所希望的，范围分解运算符可以避免这种情况。例如，假设有一个结构，它的局域标识符为 A，希望在成员函数内选全局标识符 A。编译器会缺省地选择局域的那一个，所以必须另外告诉编译器。希望用范围分解指定一个全局名字时，应当使用前面不带任何东西的运算符。这里有一个例子，表明对变量和函数的全局范围分解。

```
//: SCOPERES.CPP -- Global scope resolution
int A;
void f() {}

struct S {
  int A;
  void f();
};

void S::f() {
  ::f();  // Would be recursive otherwise!
  ::A++;  // Select the global A
  A--;    // The A at struct scope
```

```
}

main() {}
```

如果在 S::f( ) 中没有范围分解，编译器会缺省地选择 f( ) 和 A 的成员版本。

2.11 小结

在这一章中，我们已经学会了使用C++ 的基本方法，也就是在结构的内部放入函数。这种新类型被称为抽象数据类型，用这种结构创建的变量被称为这个类型的对象或实例。向对象调用成员函数被称为向这个对象发消息。面向对象的程序设计中的主要活动就是向对象发消息。

虽然将数据和函数捆绑在一起很有好处，并使得库更容易使用，因为这可以通过隐藏名字防止名字冲突，但是，还有大量的工作可以使 C++程序设计更安全。在下面一章中，将学习如何保护 struct 的一些成员，以使得只有我们能对它们操作。这就在什么是结构的用户可以改动的和什么只是程序员可以改动的之间形成了明确的界线。

2.12 练习

1) 创建一个 struct 声明，它有单个成员函数，然后对这个成员函数创建定义。创建这个新数据类型的对象，再调用这个成员函数。

2) 编写并且编译一段代码，这段代码完成数据成员选择和函数调用。

3) 写一个在另一个结构中的被声明结构的例子（嵌套结构）。并说明如何定义这个结构的成员。

4) 结构有多大？写一段能打印各个结构的长度的代码。创建一些结构，它们只有数据成员，另一些有数据成员和函数成员。然后创建一个结构，它根本没有成员。打印出所有这些结构的长度。对于根本没有成员的结构的结果作出解释。

5) C++对于枚举、联合和struct 自动创建 typedef 的等价物，正如在本章中看到的。写一个能说明这一点的小程序。

第3章　隐 藏 实 现

一个典型的C语言库通常包含一个结构和一组运行于该结构之上的相关函数。前面我们已经看到C++是怎样处理那些在概念上和语法上相关联的函数的，那就是：

把函数的声明放在一个struct内，改变这些函数的调用方法，在调用过程中不再把 struct的地址作为第一个参数传递，在程序中增加一个新的数据类型（这样就不必在 struct关键字前加一个typedef之类的声明了）。

这样做带来很多方便——有助于组织代码，使程序易于编写和阅读。然而，在使得 C++库比以前更容易的同时，存在一些其他问题，特别是在安全与控制方面。本章重点讨论 struct中的边界问题。

3.1　设置限制

在任何关系中，存在相关各方都遵从的边界是很重要的。当我们建立了一个库之后，我们就与该库的用户（也可以叫用户程序员）建立了一种关系，他是另外的程序员，但他需要用我们的库来编写一个应用程序或用我们的库来建立更大的库。

在C语言中，struct同其他数据结构一样，没有任何规则，用户可以在 struct中做他们想做的任何事情，没有办法来强制任何特殊的行为。比如，即使我们已经看到了上一章中提到的initialize()函数和cleanup()函数的重要性，但用户有权决定是否调用它们（我们将在下一章看到更好的方法）。再比如，我们可能不愿意让用户去直接处理 struct中的某些成员，但在C语言中没有任何方法可以阻止用户。一切都是暴露无遗的。

需要控制对结构成员的存取有两个理由：一是让用户避开一些他们不需要使用的工具，这些工具对数据类型内部的处理来说是必须的，但对用户特定问题的接口来说却不是必须的。这实际上是为用户提供了方便，因为他们可以很容易地知道，对他们来说哪些是重要的，哪些是可以忽略的。

二是设计者可以改变 struct的内部实现，而不必担心对用户程序员产生影响。在上一章stack的例子中，我们想以大块的方式来分配存储空间，提高速度，而不是在每次增加成员时调用malloc() 函数来重新分配内存。如果这些库的接口部分与实现部分是清楚地分开的，并作了保护，那么我们只需要让用户重新连接一遍就可以了。

3.2　C++的存取控制

C++语言引进了三个新的关键字，用于在 struct中设置边界：public、private和protected。它们的使用和含义从字面上就能理解。这些存取指定符只在 struct声明中使用，它们可以改变在它们之后的所有声明的边界。使用存取指定符，后面必须跟上一个冒号。

public意味着在其后声明的所有成员对所有的人都可以存取。 public成员就如同一般的struct成员。比如，下面的struct声明是相同的：

```
//: PUBLIC.CPP -- Public is just like C struct

struct A {
```

```
  int i;
  char j;
  float f;
  void foo();
};

void A::foo() {}

struct B {
public:
  int i;
  char j;
  float f;
  void foo();
};

void B::foo() {}
main() {}
```

private关键字则意味着，除了该类型的创建者和类的内部成员函数之外，任何人都不能存取这些成员。private在设计者与用户之间筑起了一道墙。如果有人试图存取一个私有成员，就会产生一个编译错误。在上面的例子中，我们可能想让 struct B中的部分数据成员隐藏起来，只有我们自己能存取它们：

```
//: PRIVATE.CPP -- Setting the boundary

struct B {
private:
  char j;
  float f;
public:
  int i;
  void foo();
};

void B::foo() {
  i = 0;
  j = '0';
  f = 0.0;
};

main() {
  B b;
  b.i = 1;     // OK, public
//!  b.j = '1';  // Illegal, private
//!  b.f = 1.0;  // Illegal, private
}
```

虽然foo()函数可以访问B的所有成员，但一般的全局函数如main()却不能，当然其他struct中的成员函数同样也不能。只有那些在这个struct中明确声明了的函数才能访问这些私有成员。

对存取指定符的顺序没有特别的要求，它们可以不止一次出现，它们影响在它们之后和下一个存取指定符之前声明的所有成员。

保护(protected)

最后一种存取指定符是protected。protected与private基本相似，只有一点不同：继承的结构可以访问protected成员，但不能访问private成员。但我们要到第13章才讨论继承。现在就把这两种指定符当成一样来看待，直到介绍了继承后再区分这两类指定符。

3.3 友元

如果程序员想允许不属于当前结构的一个成员函数存取结构中的数据，那该怎么办呢？他可以在struct内部声明这个函数为友元。注意，一个友元必须在一个 struct内声明，这一点很重要，因为他（和编译器）必须能读取这个结构的声明以理解这个数据类型的大小、行为等方面的规则。有一条规则在任何关系中都很重要，那就是“ 谁可以访问我的私有实现部分 ”。

类控制着哪些代码可以存取它的成员。让程序员没有办法“ 破门而入 ”，他不能声明一个新类然后说“ 嘿，我是鲍勃的朋友 ”，不能指望这样就可以访问鲍勃的私有成员和保护成员。

程序员可以把一个全局函数声明为友元类，也可以把另一个 struct中的成员函数甚至整个struct都声明为友元类，请看下面的例子：

```
//: FRIEND.CPP -- Friend allows special access

struct X; // Declaration (incomplete type spec)

struct Y {
  void f(X*);
};

struct X { // Definition
private:
  int i;
public:
  void initialize();
  friend void g(X*, int); // Global friend
  friend void Y::f(X*);  // Struct member friend
  friend struct Z; // Entire struct is a friend
  friend void h();
};

void X::initialize() { i = 0; }

void g(X* x, int i) { x->i = i; }

void Y::f(X* x) { x->i = 47; }
```

```
struct Z {
private:
  int j;
public:
  void initialize();
  void g(X* x);
};

void Z::initialize() { j = 99; }

void Z::g(X* x) { x->i += j; }

void h() {
  X x;
  x.i = 100; // Direct data manipulation
}

main() {
  X x;
  Z z;
  z.g(&x);
}
```

struct Y有一个成员函数f( )，它将修改X类型的对象。这里有一个难题，因为C++的编译器要求在引用任一变量之前必须声明，所以struct Y必须在它的成员Y :: f(X*)被声明为struct X的一个友元之前声明，但Y :: f(X*)要被声明，struct X又必须先声明。

解决的办法是：注意到Y :: f(X*)引用了一个X对象的地址。这一点很关键，因为编译器知道如何传递一个地址，这一地址大小是一定的，而不管被传递的对象类型大小。如果试图传递整个对象，编译器就必须知道X的全部定义以确定它的大小以及如何传递它，这就使程序员无法声明一个类似于Y :: g(X)的函数。

通过传递X的地址，编译器允许程序员在声明Y :: f(X*)之前做一个不完全的类型指定。这一点是在struct X的声明时完成的，这儿仅仅是告诉编译器，有一个叫X的struct,所以当它被引用时不会产生错误，只要程序员的引用不涉及名字以外的其他信息。

这样，在struct X中，Y :: f(X*)就可以成功地声明为一个友元函数，如果程序员在编译器获得对Y的全部指定信息之前声明它，就会产生一条错误，这种安全措施保证了数据的一致性，同时减少了错误的发生。

再来看看其他两个友元函数，第一个声明将一个全局函数g( )作为一个友元，但g( )在这之前并没有在全局范围内作过声明，这表明friend可以在声明函数的同时又将它作为struct的友元。这种声明对整个struct同样有效:friend struct Z是一个不完全的类型说明，并把整个struct都当作一个友元。

3.3.1 嵌套友元

一个嵌套的struct并不能自动地获得存取私有成员的权限。要获得存取私有成员的权限，必须遵守特定的规则：首先声明一个嵌套的 struct，然后声明它是全局范围使用的一个友元。

struct的声明必须与friend声明分开，否则编译器将不把它看作成员。请看下面的例子：

```
//: NESTFRND.CPP -- Nested friends
#include <stdio.h>
#include <string.h> // memset()
#define SZ 20

struct holder {
private:
  int a[SZ];
public:
  void initialize();
  struct pointer {
  private:
    holder* h;
    int* p;
  public:
    void initialize(holder* H);
    // Move around in the array:
    void next();
    void previous();
    void top();
    void end();
    // Access values:
    int read();
    void set(int i);
  };
  friend holder::pointer;
};

void holder::initialize() {
 memset(a, 0, SZ * sizeof(int));
}

void holder::pointer::initialize(holder* H) {
  h = H;
  p = h->a;
}

void holder::pointer::next() {
  if(p < &(h->a[SZ - 1])) p++;
}

void holder::pointer::previous() {
  if(p > &(h->a[0])) p--;
}
```

```
void holder::pointer::top() {
  p = &(h->a[0]);
}

void holder::pointer::end() {
  p = &(h->a[SZ - 1]);
}

int holder::pointer::read() {
  return *p;
}

void holder::pointer::set(int i) {
  *p = i;
}

main() {
  holder h;
  holder::pointer hp, hp2;
  int i;

  h.initialize();
  hp.initialize(&h);
  hp2.initialize(&h);
  for(i = 0; i < SZ; i++) {
    hp.set(i);
    hp.next();
  }
  hp.top();
  hp2.end();
  for(i = 0; i < SZ; i++) {
    printf("hp = %d, hp2 = %d\n",
           hp.read(), hp2.read());
    hp.next();
    hp2.previous();
  }
}
```

struct holder包含一个整型数组和一个pointer,我们可以通过pointer来存取这些整数。因为pointer与holder紧密相连，所以有必要将它作为struct中的一个成员。一旦pointer被定义，它就可以通过下面的声明来获得存取holder的私有成员的权限：

```
friend  holder : :pointer ;
```

注意，这里struct关键字并不是必须的，因为编译器已经知道pointer是什么了。

因为pointer是同holder分开的，所以程序员可以在main( )块中定义它的多个实例，然后用它们来选择数组的不同部分。由于pointer是C语言中指针的替代，因此程序员可以保证它总是安全地指向holder的内部。

3.3.2 它是纯的吗

这种类的定义提供了有关权限的信息，我们可以知道哪些函数可以改变类的私有部分。如果一个函数被声明为friend，就意味着它不是这个类的成员函数，但却可以修改类的私有成员，而且它必须被列在类的定义中，因此我们可以认为它是一个特权函数。

C++不是完全的面向对象语言，它只是一个混合产品。 friend关键字就是用来解决部分的突发问题。它也说明了这种语言是不纯的。毕竟 C++语言的设计是为了实用，而不是追求理想的抽象。

3.4 对象布局

第2章讲述了为C编译器而写的一个struct，然后一字不动地用C++编译器进行编译。这里我们就来分析struct的布局，也就是，各自的变量放在内存的什么位置？如果C++编译器改变了C struct中的布局，在C语言代码中如果使用了struct中变量的位置信息的话，那么在C++中就会出错。

当我们开始使用一个存取指定符时，我们就已经完全进入了 C++的世界，情况开始有所改变。在一个特定的“ 存取块 ”（被存取指定符限定的一组声明）内，这些变量在内存中肯定是相邻的，这和C语言中一样，然而这些“ 存取块 ”本身可以不按定义的顺序在对象中出现。

虽然编译器通常都是按存取块出现的顺序给它们分配内存，但并不是一定要这样，因为部分机器的结构或操作环境可对私有成员和保护成员提供明确的支持，将其放在特定的内存位置上。C++语言的存取指定并不想限制这种好处。

存取指定符是struct的一部分，它并不影响从这个 struct产生的对象，程序开始运行时，所有的存取指定信息都消失了。存取指定信息通常是在编译期间消失的。在程序运行期间，对象变成了一个存储区域，别无他物，因此，如果有人真的想破坏这些规则并且直接存取内存中的数据，就如在C中所做的那样，那么C++并不能防止他做这种不明智的事，它只是提供给人们一个更容易、更方便的方法。

一般说来，程序员写程序时，依赖特定实现的任何东西都是不合适的。如确有必要，这些指定应封装在一个struct之内，这样当环境改变时，他只需修改一个地方就行了。

3.5 类

存取控制通常是指实现细节的隐藏。将函数包含到一个 struct内（封装）来产生一种带数据和操作的数据类型，但由存取控制在该数据类型之内确定边界。这样做的原因有两个：首先是决定哪些用户可以用，哪些用户不能用。我们可以建立内部的数据结构，而用户只能用接口部分的数据，我们不必担心用户会把内部的数据当作接口数据来存取。

这就直接导出第二个原因，那就是将具体实现与接口分离开来。如果该结构被用在一系列的程序中，而用户只是对公共的接口发送消息，这样程序员就可以改变所有声明为 private的成员而不必去修改用户的代码。

封装和实现细节的隐藏能防止一些情况的发生，而这在 C语言的struct类型中是做不到的。我们现在已经处在面向对象编程的世界中，在这里，结构就是一个对象的类，就像人们可以描述一个鱼类或一个鸟类，任何属于该类的对象都共享这些特征和行为。也就是说，结构的声明开始描述该类型的所有对象及其行为。

在最初的面向对象编程语言 Simula-67中，关键字class被用来描述一个新的数据类型。这显然激发了Stroustrup在C++中选用同样的关键字，以强调这是整个语言的关键所在。新的数据

类型并非只是C中的带有函数的struct，这当然需要用一个新的关键字。

然而class在C++中的使用逐渐变成了一个非必要的关键字。它和struct的每个方面都是一样的，除了class中的成员缺省为私有的，而struct中的成员缺省为public。下面有两个结构，它们将产生相同的结果。

```
//: CLASS.CPP -- Similarity of struct and class

struct A {
private:
  int i, j, k;
public:
  int f();
  void g();
};

int A::f() { return i + j + k; }

void A::g() { i = j = k = 0; }

// Identical results are produced with:

class B {
  int i, j, k;
public:
  int f();
  void g();
};

int B::f() { return i + j + k; }

void B::g() { i = j = k = 0; }

main() {}
```

在C++中，class是面向对象语言的基本概念，它是一个关键字，本书将不用粗体字来表示。由于要经常用到class，这样做很麻烦。但转换到类是如此重要，我怀疑Stroustrup偏向于将struct重新定义，但考虑到向后兼容性而没有这样做。

许多人喜欢用一种更像struct的风格去创建一个类，因为可以通过以public开头来重载“缺省为私有”的类行为。

```
class X {
public:
  void interface_function();
private:
  void private_function();
  int internal_representation;
};
```

之所以这样做，是因为这样可以让读者更清楚地看到他们的成员是与什么限定符相连的，这样他们可以忽略所有声明为私有成员。事实上，所有其他成员都必须在类中声明的原因仅仅是让编译器知道对象有多大，以便为它们分配合适的存储空间，并保证它们的一致性。

但本书中仍采用首先声明私有成员的方法，如下例：

```
class X {
  void private_function();
  int internal_representation;
public:
  void interface_function();
};
```

有些人甚至不厌其烦地在他们的私有成员名字前加上私有标志：

```
class Y {
public:
  void f();
private:
  int mX;  // "self-mangled" name
};
```

因为mX已经隐藏于Y的范围内，所以在x之前加m并不是必须的。然而在一个有许多全局变量的项目中（有些事虽然我们想极力避免，但有时仍不可避免地出现），它有助于在一个成员函数的定义体内识别出哪些是全局变量，哪些是成员变量。

3.5.1 用存取控制来修改stash

现在我们把第2章的例子用类及存取控制来改写一下。请注意用户的接口部分现在已经很清楚地区分开了，完全不用担心用户会偶然地访问他们不该访问的内容了。

```
//: STASH.H -- Converted to use access control
#ifndef STASH_H_
#define STASH_H_
class stash {
  int size;  // Size of each space
  int quantity; // Number of storage spaces
  int next; // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char* storage;
  void inflate(int increase);
public:
  void initialize(int Size);
  void cleanup();
  int add(void* element);
  void* fetch(int index);
  int count();
};
#endif // STASH_H_
```

inflate()函数声明为私有，因为它只被add()函数调用，所以它属于内在实现部分，不属于接口部分。这就意味着以后我们可以调整这些实现的细节，用不同的系统来管理内存。在此例中除了包含文件的名字之外，只有上面的头文件需要更改，实现文件和测试文件是相同的。

3.5.2 用存取控制来修改stack

对于第二个例子，我们把stack改写成一个类。现在嵌套的数据结构是私有的。这样做的好处是可以确保用户既看不到它，也不能依赖stack的内部表示：

```
//: STACK.H -- Nested structs via linked list
#ifndef STACK_H_
#define STACK_H_

class stack {
  struct link {
    void* data;
    link* next;
    void initialize(void* Data, link* Next);
  } * head;
public:
  void initialize();
  void push(void* Data);
  void* peek();
  void* pop();
  void cleanup();
};
#endif // STACK_H_
```

和上例一样，实现部分无需改动，这里不再赘述。测试部分也一样，唯一改动的地方是类的接口部分的健壮性。存取控制的真正价值体现在开发阶段，防止越界。事实上，只有编译器知道类成员的保护级别，并没有将此类的信息传递给连接器。所有的存取保护检查都是由编译器来完成的，在运行期间不再检查。

注意面向用户的接口部分现在是一个压入堆栈。它是用一个链表结构来实现的，但可以换成其他的形式，而不会影响用户处理问题，更重要的是，无需改动用户的代码。

3.6 句柄类（handle classes）

C++中的存取控制允许将实现与接口部分分开，但实现的隐藏是不完全的。编译器必须知道一个对象的所有部分的声明，以便创建和管理它。我们可以想象一种只需声明一个对象的公共接口部分的编程语言，而将私有的实现部分隐藏起来。但 C++在编译期间要尽可能多地做静态类型检查。这意味着尽早捕获错误，也意味着程序具有更高的效率。然而这对私有的实现部分来说带来两个影响：一是即使程序员不能轻易地访问实现部分，但他可以看到它；二是造成一些不必要的重复编译。

3.6.1 可见的实现部分

有些项目不可让最终用户看到其实现部分。例如可能在一个库的头文件中显示一些策略

信息，但公司不想让这些信息被竞争对手获得。比如从事一个安全性很重要的系统 (如加密算法)，我们不想在文件中暴露任何线索，以防有人破译我们的代码。或许我们把库放在了一个“有敌意”的环境中，在那里程序员会不顾一切地用指针和类型转换存取我们的私有成员。在所有这些情况下，就有必要把一个编译好的实际结构放在实现文件中，而不是让其暴露在头文件中。

3.6.2 减少重复编译

在我们的编程环境中，当一个文件被修改，或它所依赖的文件包含的头文件被修改时，项目负责人需要重复编译这些文件。这意味着无论何时程序员修改了一个类，无论是修改公共的接口部分，还是私有的实现部分，他都得再次编译包含头文件的所有文件。对于一个大的项目而言，在开发初期这可能非常难以处理，因为实现部分可能需要经常改动；如果这个项目非常大，用于编译的时间过多就可能妨碍项目的完成。

解决这个问题的技术有时叫句柄类（handle classes）或叫“Cheshire Cat”[1]。有关实现的任何东西都消失了，只剩一个单一的指针“smile”。该指针指向一个结构，该结构的定义与其所有的成员函数的定义一样出现在实现文件中。这样，只要接口部分不改变，头文件就不需变动。而实现部分可以按需要任意更动，完成后只要对实现文件进行重新编译，然后再连接到项目中。

这里有个说明这一技术的简单例子。头文件中只包含公共的接口和一个简单的没有完全指定的类指针。

```
//: HANDLE.H -- Handle classes
#ifndef HANDLE_H_
#define HANDLE_H_

class handle {
  struct cheshire; // Class declaration only
  cheshire* smile;
public:
  void initialize();
  void cleanup();
  int read();
  void change(int);
};
#endif // HANDLE_H_
```

这是所有客户程序员都能看到的。这行

```
struct cheshire;
```

是一个没有完全指定的类型说明或类声明（一个类的定义包含类的主体）。它告诉编译器，cheshire 是一个结构的名字，但没有提供有关该结构的任何东西。这对产生一个指向结构的指针来说已经足够了。但我们在提供一个结构的主体部分之前不能创建一个对象。在这种技术里，包含具体实现的结构主体被隐藏在实现文件中。

---

[1] 这个名字是归属于John Carolan和Lewis Carroll，前者是C++最早的开创者之一。

```
//: HANDLE.CPP -- Handle implementation
#include "..\2\handle.h"
#include <stdlib.h>
#include <assert.h>

// Define handle's implementation:
struct handle::cheshire {
  int i;
};

void handle::initialize() {
  smile = (cheshire*)malloc(sizeof(cheshire));
  assert(smile);
  smile->i = 0;
}

void handle::cleanup() {
  free(smile);
}

int handle::read() {
  return smile->i;
}

void handle::change(int x) {
  smile->i = x;
}
```

cheshire 是一个嵌套结构，所以它必须用范围分解符定义

```
struct handle::cheshire {
```

在handle::initialize()中，为cheshire struct分配存储空间 [1]，在handle::cleanup()中这些空间被释放。这些内存被用来代替类的所有私有部分。当编译 HANDLE.CPP时，这个结构的定义被隐藏在目标文件中，没有人能看到它。如果改变了 cheshire的组成，唯一要重新编译的是HANDLE.CPP，因为头文件并没有改动。

句柄（handle）的使用就像任何类的使用一样，包含头文件、创建对象、发送信息。

```
//: USEHANDL.CPP -- Use the handle class
#include "..\2\handle.h"

main() {
  handle u;
  u.initialize();
  u.read();
```

[1] 在第12章我们将看到创建对象更好的方法：用new在堆中分配内存。

```
  u.change(1);
  u.cleanup();
}
```

客户程序员唯一能存取的就是公共的接口部分，因此，只是修改了在实现中的部分，这些文件就不须重新编译。虽然这并不是完美的信息隐藏，但毕竟是一大进步。

3.7 小结

在C++中，存取控制并不是面向对象的特征，但它为类的创建者提供了很有价值的访问控制。类的用户可以清楚地看到，什么可以用，什么应该忽略。更重要的是，它保证了类的用户不会依赖任何类的实现细节。有了这些，我们就能更改类的实现部分，没有人会因此而受到影响，因为他们并不能访问类的这一部分。

一旦我们有了更改实现部分的自由，就可以在以后的时间里改进我们的设计，而且允许犯错误。要知道，无论我们如何小心地计划和设计，都可能犯错误。知道犯些错误也是相对安全的，这意味着我们会变得更有经验，会学得更快，就会更早完成项目。

一个类的公共接口部分是用户能看到的。所以在分析设计阶段，保证接口的正确性更加重要。但这并不是说接口不能作修改。如果我们第一次没有正确地设计接口部分，我们可以再增加函数，这样就不需要删除那些已使用该类的程序代码。

3.8 练习

1) 创建一个类，具有public、private 和protected数据成员和函数成员。创建该类的一个对象，看看当试图存取所有的类成员时会得到一些什么编译信息。

2) 创建一个类和一个全局friend函数来处理类的私有数据。

3) 修改 HANDLE.CPP中的 cheshire，重新编译和连接这一文件，但不重新编译USEHANDL.CPP。

第4章　初始化与清除

第2章利用了一些分散的典型C语言库的构件，并把它们封装在一个struct中，从而在库的应用方面做了有意义的改进。（从现在起，这个抽象数据类型称为类）。

这样不仅为库构件提供了单一一致的入口指针，也用类名隐藏了类内部的函数名。在第3章中，我们介绍了存取控制（隐藏实现），这就为类的设计者提供了一种设立界线的途径，通过界线的设立来决定哪些是用户可以处理的，哪些是禁止的。这意味着数据类型的内部机制对设计者来说是可控的和能自行处理的。这样让用户也清楚哪些成员是他们能够使用并加以注意的。

封装和实现的隐藏大大地改善了库的使用。它们提供的新的数据类型的概念在某些方面比从C中继承的嵌入式数据类型要好。现在C++编译器可以为这种新的数据类型提供类型检查，这样在使用这种数据类型时就确保了一定的安全性。

当然，说到安全性，C++的编译器能比C编译器提供更多的功能。在本章及以后的章节中，我们将看到许多C++的另外一些性能。它们可以让我们程序中的错误暴露无遗，有时甚至在我们编译这个程序之前，帮我们查出错误，但通常是编译器的警告和出错信息。所以我们不久就会习惯：在第一次编译时总听不到编译器那意味着正确的提示音。

安全性包括初始化和清除两个方面。在C语言中，如果程序员忘记了初始化或清除一个变量，就会导致一大段程序错误。这在一个库中尤其如此，特别是当用户不知如何对一个struct初始化，甚至不知道必须要初始化时。（库中通常不包含初始化函数，所以用户不得不手工初始化struct）。清除是一个特殊问题，因为C程序员一旦用过了一个变量后就把它忘记了，所以对一个库的struct来说，必要的清除工作往往被遗忘了。

在C++中，初始化和清除的概念是简化类库使用的关键所在，并可以减少那些由于用户忘记这些操作而引起的许多细微错误。本章就来讨论C++的这些特征。

4.1　用构造函数确保初始化

在stash和stack类中都曾调用initialize()函数，这暗示无论用什么方法使用这些类的对象，在使用之前都应当调用这一函数。很不幸的是，这要求用户必须正确地初始化。而用户在专注于用那令人惊奇的库来解决他们的问题的时候，往往忽视了这些细节。在C++中，初始化实在太重要了，所以不能留给用户来完成。类的设计者可以通过提供一个叫做构造函数的特殊函数来保证每个对象都正确的初始化。如果一个类有构造函数，编译器在创建对象时就自动调用这一函数，这一切在用户使用他们的对象之前就已经完成了。对用户来说，是否调用构造函数并不是可选的，它是由编译器在对象定义时完成的。

接下来的问题是这个函数叫什么名字。这必须考虑两点，首先这个名字不能与类的其他成员函数冲突，其次，因为该函数是由编译器调用的，所以编译器必须总能知道调用哪个函数。Stroustrup的方法似乎是最容易也是最符合逻辑的：构造函数的名字与类的名字一样。这使得这样的函数在初始化时自动被调用。

下面是一个带构造函数的类的简单例子：

```
class X {
  int i;
public:
  X();  // constructor
};
```

现在当一个对象被定义时：

```
void f() {
  X a;
  // ...
}
```

这时就好像a是一个整数一样：为这个对象分配内存。但是当程序执行到 a的定义点时，构造函数自动被调用，因为编译器已悄悄地在 a的定义点处插入了一个 X::X()的调用。就像其他成员函数被调用一样。传递到构造函数的第一个参数（隐含）是调用这一函数对象的地址。

像其他函数一样，我们也可以通过构造函数传递参数，指定对象该如何创建，设定对象初始值等等。构造函数的参数保证对象的所有部分都被初始化成合适的值。举例来说：如果类tree有一个带整型参数的构造函数，用以指定树的高度，那么我们就必须这样来创建一个对象：

```
tree  t(12);  // 12英尺高的树
```

如果tree(int)是唯一的构造函数，编译器将不会用其他方法来创建一个对象（在下一章我们将看到多个构造函数以及调用它们的不同方法）。

关于构造函数，我们就全部介绍完了。构造函数是一个有着特殊名字，由编译器自动为每个对象调用的函数，然而它解决了类的很多问题，并使得代码更容易阅读。例如在上一个代码段中，对有些initialize()函数我们并没有看到显式的调用，这些函数从概念上说是与定义分开的。在C++中，定义和初始化是同一概念，不能只取其中之一。

构造函数和析构函数是两个非常特殊的函数：它们没有返回值。这与返回值为 void的函数显然不同。后者虽然也不返回任何值，但我们还可以让它做点别的。而构造函数和析构函数则不允许。在程序中创建和消除一个对象的行为非常特殊，就像出生和死亡，而且总是由编译器来调用这些函数以确保它们被执行。如果它们有返回值，要么编译器必须知道如何处理返回值，要么就只能由用户自己来显式地调用构造函数与析构函数，这样一来，安全性就被破坏了。

4.2 用析构函数确保清除

作为一个C程序员，我们可能经常想到初始化的重要性，但很少想到清除的重要性。毕竟，清除一个整型变量时需要作什么 ?只需要忘记它。然而，在一个库中，对于一个曾经用过的对象，仅仅“忘记它”是不安全的。如果它修改了某些硬件参数，或者在屏幕上显示了一些字符，或在堆中分配了一些内存，那么将会发生什么呢？如果我们只是“忘记它”，我们的对象就永远不会消失。在C++中，清除就像初始化一样重要。通过析构函数来保证清除的执行。

析构函数的语法与构造函数一样，用类的名字作函数名。然而析构函数前面加上一个 ~，以和构造函数区别。另外，析构函数不带任何参数，因为析构不需任何选项。下面是一个析构函数的声明：

```
class Y {
public:
  ~Y();
};
```

当对象超出它的定义范围时，编译器自动调用析构函数。我们可以看到，在对象的定义点处构造函数被调用，但析构函数调用的唯一根据是包含该对象的右括号，即使用 goto语句跳出这一程序块（为了与C 语言向后兼容，goto在C++中仍然存在，当然也是为了方便）。我们应该注意一些非本地的goto语句，它们用标准C语言库中的setjmp() 和longjmp()函数，这些函数将不会引发析构函数的调用。（这里作一点说明：有的编译器可能并不用这种方法来实现。依赖那些不在说明书中的特征意味着这样的代码是不可移植的）。

下例说明了构造函数与析构函数的上述特征：

```
//: CONSTR1.CPP -- Constructors & destructors
#include <stdio.h>

class tree {
  int height;
public:
  tree(int initialHeight);  // Constructor
  ~tree();  // Destructor
  void grow(int years);
  void printsize();
};

tree::tree(int initialHeight) {
  height = initialHeight;
}

tree::~tree() {
  puts("inside tree destructor");
  printsize();
}

void tree::grow(int years) {
  height += years;
}

void tree::printsize() {
  printf("tree height is %d\n", height);
}

main() {
  puts("before opening brace");
  {
    tree t(12);
    puts("after tree creation");
    t.printsize();
    t.grow(4);
    puts("before closing brace");
```

```
  }
  puts("after closing brace");
}
```

下面是上面程序的输出结果：

```
before opening brace
after tree creation
tree height is 12
before closing brace
inside tree destructor
tree height is 16
after closing brace
```

我们可以看到析构函数在包括它的右括号处被调用。

4.3 清除定义块

在C中，我们总要在一个程序块的左括号一开始就定义好所有的变量，这在程序设计语言中不算少见（Pascal中例外），其理由无非是因为“这是一个好的编程风格”。在这点上，我有自己的看法。我认为它总是给我带来不便。作为一个程序员，每当我需要增加一个变量时我都得跳到块的开始，我发现如果变量定义紧靠着变量的使用处时，程序的可读性更强。

也许这些争论具有一定的普遍性。在C++中，是否一定要在块的开头就定义所有变量成了一个很突出的问题。如果存在构造函数，那么当对象产生时它必须首先被调用，如果构造函数带有一个或者更多个初始化参数，我们怎么知道在块的开头定义这些初始化信息呢？在一般的编程情况下，我们做不到这点，因为C中没有私有成员的概念。这样很容易将定义与初始化部分分开，然而C++要保证在一个对象产生时，它同时被初始化。这可以保证我们的系统中没有未初始化的对象。C并不关心这些。事实上，C要求我们在块的开头就定义所有变量，在我们还不知道一些必要的初始化信息时，就要求我们这样做是鼓励我们不初始化变量。

通常，在C++中，在还不拥有构造函数的初始化信息时不能创建一个对象，所以不必在块的开头定义所有变量。事实上，这种语言风格似乎鼓励我们把对象的定义放得离使用点尽可能近一点。在C++中，对一个对象适用的所有规则，对预定义类型也同样适用。这意味着任何类的对象或者预定义类型都可以在块的任何地点定义。这也意味着我们可以等到我们已经知道一个变量的必要信息时再去定义它，所以我们总是可以同时定义和初始化一个变量。

```
//: DEFINIT.CPP -- Defining variables anywhere
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>

class G {
  int i;
public:
  G(int I);
};

G::G(int I) { i = I; }
```

```
main() {
  #define SZ 100
  char buf[SZ];
  printf("initialization value? ");
  int retval = (int)gets(buf);
  assert(retval);
  int x = atoi(buf);
  int y = x + 3;
  G g(y);
}
```

我们可以看到首先是buf被定义，然后是一些语句，然后x被定义并用一个函数调用对它初始化，然后y和g被定义。在C中这些变量都只能在块的一开始定义。一般说来，应该在尽可能靠近变量的使用点定义变量，并在定义时就初始化（这是对预定义类型的一种建议，但在那里可以不做初始化）。这是出于安全性的考虑，减少变量误用的可能性。另外，程序的可读性也增强了，因为读者不需要跳到程序头去确定变量的类型。

4.3.1 for循环

在C++中，我们将经常看到for循环的计数器直接在for表达式中定义：

```
for(int j = 0; j < 100; j++) {
    printf("j = %d\n", j);
}
for(int i = 0; i < 100; i++)
    printf("i = %d\n", i);
```

上述声明是一种重要的特殊情况，这可能使那些刚接触C++的程序员感到迷惑不解。

变量i和j都是在for表达式中直接定义的（在C中我们不能这样做），然后他们就作为一个变量在for循环中使用。这给程序员带来很大的方便，因为从上下文中我们可以清楚地知道变量 i、j的作用，所以不必再用诸如i_loop_counter之类的名字来定义一个变量，以表示这一变量的作用。

这里有一个变量生存期的问题，在以前这是由程序块的右大括号来确定的。从编译器的角度来看这样是合理的，因为作为程序员，我们显然想让 i只在循环内部有效。然而很不幸的是，如果我们用这种方法声明：

```
for(int i = 0; i < 100; i++)
    printf("i = %d\n", i);
// ....
for(int i = 0; i < 100; i++){
    printf("i = %d\n", i);
}
```

（无论有没有大括号，）在同一程序块内，编译器将给出一个重复定义的错误，而新的标准C++说明书上说，一个在for循环的控制表达式中定义的循环计数器只在该循环内才有效，所以上面的声明是可行的。（当然，并不是所有的编译器都支持这一点，我们可能会遇到一些老式风格的编译器。）如果这种转变引起一些错误的话，编译器会指出，解决起来也很容易。注意，

那些局部变量会屏蔽这个封闭范围中的变量。

我发现一个在小范围内设计良好的指示器：如果我们的一个函数有好几页，也许我们正在试图让这个函数完成太多的工作。用更多的细化的函数不仅有用，而且更容易发现错误。

4.3.2 空间分配

现在，一个变量可以在某个程序范围内的任何地方定义，所以在这个变量的定义之前是无法对它分配内存空间的。通常，编译器更可能像C编译器一样，在一个程序块的开头就分配所有的内存。这些对我们来说是无关紧要的，因为作为一个程序员，我们在变量定义之前总是无法得到存储空间。即使存储空间在块的一开始就被分配，构造函数也仍然要到对象的定义时才会被调用，因为标识符只有到此时才有效。编译器甚至会检查我们有没有把一个对象的定义放到一个条件块中，比如在switch块中声明，或可能被goto跳过的地方。

下例中解除注释的句子会导致一个警告或一个错误。

```
//: NOJUMP.CPP -- Can't jump past constructors

class X {
public:
  X() {}
};

void f(int i) {
  if(i < 10) {
   //! goto jump1; // Error: goto bypasses init
  }
  X x1;  // Constructor called here
 jump1:
  switch(i) {
    case 1 :
      X x2;  // Constructor called here
      break;
  //! case 2 : // Error: case bypasses init
      X x3;  // Constructor called here
      break;
  }
}

main() {}
```

在上面的代码中，goto和switch都可能跳过构造函数的调用点，然而这个对象会在后面的程序块中起作用，这样，构造函数就没有被调用，所以编译器给出了一条出错信息。这就确保了对象在产生的同时被初始化。

当然，这里讨论的内存分配都是在一个堆栈中。内存分配是通过编译器向下移动堆栈指针来实现的（这只是相对而言，实际指针值可能增加，也可能减少，这依赖于机器）。也可以在堆中分配对象的内存，这将在第12章中介绍。

4.4 含有构造函数和析构函数的stash

在前几章的例子中，都有一些很明显的函数对应为构造函数和析构函数： initialize()和cleanup()。下面是带有构造函数与析构函数的stash头文件。

```
//: STASH3.H -- With constructors & destructors
#ifndef STASH3_H_
#define STASH3_H_

class stash {
  int size;  // Size of each space
  int quantity; // Number of storage spaces
  int next; // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char* storage;
  void inflate(int increase);
public:
  stash(int Size);
  ~stash();
  int add(void* element);
  void* fetch(int index);
  int count();
};
#endif // STASH3_H_
```

下面是实现文件，这里只对initialize()和cleanup()的定义作了修改，它们分别用构造函数与析构函数代替。

```
//: STASH3.CPP -- Constructors & destructors
#include "..\3\stash3.h"
#include <assert.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

stash::stash(int Size) {
  size = Size;
  quantity = 0;
  storage = 0;
  next = 0;
}

stash::~stash() {
  if(storage) {
    puts("freeing storage");
    free(storage);
  }
```

```
}

int stash::add(void* element) {
  if(next >= quantity) // Enough space left?
    inflate(100);
  // Copy element into storage,
  // starting at next empty space:
  memcpy(&(storage[next * size]),
         element, size);
  next++;
  return(next - 1); // Index number
}

void* stash::fetch(int index) {
  if(index >= next || index < 0)
    return 0;  // Not out of bounds?
  // Produce pointer to desired element:
  return &(storage[index * size]);
}
int stash::count() {
  return next; // Number of elements in stash
}

void stash::inflate(int increase) {
  void* v =
    realloc(storage, (quantity+increase)*size);
  assert(v);  // Was it successful?
  storage = (unsigned char*)v;
  quantity += increase;
}
```

注意，在下面的测试程序中，stash对象的定义放在紧靠对象调用的地方，对象的初始化通过构造函数的参数列表来实现，而对象的初始化似乎成了对象定义的一部分。

```
//: STSHTST3.CPP -- Constructors & destructors
#include "..\3\stash3.h"
#include <stdio.h>
#include <assert.h>
#define BUFSIZE 80

main() {
  stash intStash(sizeof(int));
  for(int j = 0; j < 100; j++)
    intStash.add(&j);

  FILE* file = fopen("STASHTST.CPP", "r");
  assert(file);
```

```
// Holds 80-character strings:
stash stringStash(sizeof(char) * BUFSIZE);
char buf[BUFSIZE];
while(fgets(buf, BUFSIZE, file))
  stringStash.add(buf);
fclose(file);
  for(int k = 0; k < intStash.count(); k++)
    printf("intStash.fetch(%d) = %d\n", k,
           *(int*)intStash.fetch(k));

  for(int i = 0; i < stringStash.count(); i++)
    printf("stringStash.fetch(%d) = %s",
           i, (char*)stringStash.fetch(i++));
  putchar('\n');
}
```

再看看cleanup()调用已被取消，但当intStash和stringStash越出程序块的范围时，析构函数被自动地调用了。

4.5 含有构造函数和析构函数的stack

重新实现含有构造函数和析构函数的链表（在stack内）。这是修改后的头文件：

```
//: STACK3.H -- With constructors/destructors
#ifndef STACK3_H_
#define STACK3_H_

class stack {
  struct link {
    void* data;
    link* next;
    void initialize(void* Data, link* Next);
  } * head;
public:
  stack();
  ~stack();
  void push(void* Data);
  void* peek();
  void* pop();
};
#endif // STACK3_H_
```

注意，虽然stack有构造函数与析构函数，但嵌套类link并没有，这并不是说它不需要。当它被使用时，问题就来了：

```
//: STACK3.CPP -- Constructors/destructors
#include <stdlib.h>
#include <assert.h>
```

```
#include "..\3\stack3.h"

void stack::link::initialize(
  void* Data, link* Next) {
  data = Data;
  next = Next;
}

stack::stack() { head = 0; }

void stack::push(void* Data) {
  // Can't use a constructor with malloc!
  link* newlink = (link*)malloc(sizeof(link));
  assert(newlink);
  newlink->initialize(Data, head);
  head = newlink;
}

void* stack::peek() { return head->data; }

void* stack::pop() {
  if(head == 0) return 0;
  void* result = head->data;
  link* oldHead = head;
  head = head->next;
  free(oldHead);
  return result;
}

stack::~stack() {
  link* cursor = head;
  while(head) {
    cursor = cursor->next;
    free(head->data); // Assumes malloc!
    free(head);
    head = cursor;
  }
}
```

link是在stack::push内部产生的，但它是创建在堆栈上的，这儿就产生了一个疑难问题。

如果一个对象有构造函数，我们怎么创建它呢？到目前为止，我们一直这样说：“好吧，这是堆中的一块内存，我想您就假定它是给这个对象的吧。”但构造函数并不允许我们就这样把一个内存地址交给它来创建一个对象[1]。对象的创建很关键，C++的构造函数想控制整个过

[1] 实际上，确有允许这么做的语法。但它是在特殊情况下使用的，不能解决在此描述的一般问题。

程以保证安全。有个很容易的解决办法，那就是用new操作符，我们将在第12章讨论这个问题。现在，只要用C中的动态内存分配就行了。因为内存分配与清除都隐藏在stack中，它是实现部分，我们在测试程序中看不到它的影响。

```
//: STKTST3.CPP -- Constructors/destructors
#include "..\3\stack3.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

main(int argc, char** argv) {
  assert(argc == 2); // File name is argument
  FILE* file = fopen(argv[1], "r");
  assert(file);
  #define BUFSIZE 100
  char buf[BUFSIZE];
  stack textlines;  // Constructor called here
  // Read file and store lines in the stack:
  while(fgets(buf, BUFSIZE, file)) {
    char* string =
      (char*)malloc(strlen(buf) + 1);
    assert(string);
    strcpy(string, buf);
    textlines.push(string);
  }
  // Pop lines from the stack and print them:
  char* s;
  while((s = (char*)textlines.pop()) != 0) {
    printf("%s", s); free(s); }
}  // Destructor called here
```

textlines的构造函数与析构函数都是自动调用的，所以类的用户只要把精力集中于怎样使用这些对象上，而不需要担心它们是否已被正确地初始化和清除了。

4.6 集合初始化

集合，顾名思义，就是多个事物聚集在一起。这个定义包括各种类型的集合：像 struct和class等。数组就是单一类型的集合。

初始化集合往往既冗长又容易出错。而C++中集合的初始化却变得很方便而且很安全。当我们产生一个集合对像时，我们要做的只是指定初始值就行了，然后初始化工作就由编译器去承担了。这种指定可以用几种不同的风格，取决于我们正在处理的集合类型。但不管是哪种情况，指定的初值都要用大括号括起来。比如一个预定义类型的数组可以这样定义：

```
int a[5]={1,2,3,4,5};
```

如果给出的初始化值多于数组元素的个数，编译器就会给出一条出错信息。但如果给的初

始化值少于数组元素的个数，那将会怎么样呢？例如：

```
int b[6]={0};
```

这时，编译器会把第一个初始化值赋给数组的第一个元素，然后用 0赋给其余的元素。注意，如果我们定义了一个数组而没有给出一列初始值时，编译器并不会去做这些工作。所以上面的表达式是将一个数组初始化为零的简洁方法，它不需要用一个 for循环，也避免了“偏移1位”错误（它可能比for循环更有效，这依赖于编译器）。

数组还有一种叫自动计数的快速初始化方法，就是让编译器按初始化值的个数去决定数组的大小：

```
int c[] = {1,2,3,4};
```

现在，如果我们决定增加其他的元素到这个数组上，只要增加一个初始化值即可，如果以此建立我们的代码，只需在一处作出修改即可，这样，我们在修改时出错的机会就减少了。但怎样确定这个数组的大小呢？用表达式sizeof c/sizeof *c(整个数组的大小除以第一个元素的大小)即可算出，这样，当数组大小改变时它无需修改。

```
for(int i = 0; i< sizeof c / sizeof *c; i++)
  c[i]++;
```

struct也是一种集合类型，它们也可以用同样的方式初始化。因为 C风格的struct的所有成员都是公共型的，所以它们的值可以直接指定：

```
struct X {
  int i;
  float f;
  char c;
};
X x1 = {1,2.2,'c' };
```

如果我们有一个这种struct的数组，我们也可以用嵌套的大括号来初始化每一个对象。

```
X x2[3] = {{1,1.1, 'a'},{2,2.2, 'b'}};
```

这里，第三个对象被初始化为零。

如果struct中有私有成员，或即使所有成员都是公共成员，但有一个构造函数，情况就不一样了。在上例中，初始值被直接赋给了集合中的每个元素，但构造函数是通过外在的接口来强制初始化的。这里，构造函数必须被调用来完成初始化，因此，如果有一个下面的 struct类型：

```
struct Y {
  float f;
  int i;
  Y(int A); // presumably assigned to i
};
```

我们必须指示构造函数调用，最好的方法像下面这样：

```
Y y2[] = {Y(1),Y(2),Y(3)};
```

这样我们就得到了三个对象和进行了三次构造函数调用。只要有构造函数，无论是所有成员都是公共的struct还是一个带私有成员的class,所有的初始化工作都必须通过构造函数，即使我们正在对一个集合初始化。

下面是构造函数带多个参数的又一个例子：

```
//: MULTIARG.CPP -- Multiple constructor arguments
// with aggregate initialization

class X {
  int i, j;
public:
  X(int I, int J) {
    i = I;
    j = J;
  }
};

main() {
  X xx[] = { X(1,2), X(3,4), X(5,6), X(7,8) };
}
```

注意，它看上去就像数组中的每个对象都对一个没有命名的构造函数调用了一次一样。

4.7 缺省构造函数

缺省构造函数就是不带任何参数的构造函数。缺省的构造函数用来创建一个“香草（vanilla）对象”,当编译器需要创建一个对象而又不知任何细节时，缺省的构造函数就显得非常重要。比如，我们有一个类Y，并用它来定义对象：

```
Y y4[2] = {Y(1)}
```

编译器就会报告找不到缺省的构造函数，数组中的第二个对象想不带参数来创建，所以编译器就去找缺省的构造函数。实际上，如果我们只是简单地定义了一个Y对象的数组：

```
Y y5[7] ;
```

或一个单一的对象

```
Y y;
```

编译器会报告同样的错误，因为它必须用一个缺省的构造函数去初始化数组中的每个对象。（记住，一旦有了一个构造函数，编译器就会确保不管在什么情况下它总会被调用）。

缺省的构造函数是如此重要，所以在一个构造类型（ struct 或class）中没有构造函数时，编译器会自动创建一个。因此下面例子将会正常运行：

```
class Z {
  int i; // private
}; // no constructor

Z z,z2[10];
```

然而，一旦有构造函数而没有缺省构造函数，上面的对象定义就会产生一个编译错误。

我们可能会想，缺省构造函数应该可以做一些智能化的初始化工作，比如把对象的所有内存置零。但事实并非如此。因为这样会增加额外的负担，而且使程序员无法控制。比如，如果我们把在C中编译过的代码用在C++中，就会导致不同的结果。如果我们想把内存初始化为零，必须亲自去做。

对一个C++的新手来说，自动产生的缺省构造函数并不会使编程更容易。它实际上要求与已有的C代码保持向后兼容。这是C++中的一个关键问题。在C中，创建一个struct数组的情况

很常见，而在C++中，在没有缺省构造函数时，这会引起一个编译错误。

如果我们仅仅因为风格问题就去修改我们的 C代码，然后用C++重新编译，也许我们会很不乐意。当将C代码在C++中编译时，我们总会遇到这样、那样的编译错误，但这些错误都是C++编译器所发现的C的不良代码，因为C++的规则更严格。事实上，用C++编译器去编译C代码是一个发现潜在错误的很好的方法。

4.8 小结

由C++提供的细致精巧机制应给我们这样一个强烈的暗示：在这个语言中，初始化和清除是多么至关重要。在Stroustrup设计C++时，他所作的第一个有关C语言产品的观察就是，由于没有适当地初始化变量，从而导致了程序不可移植的问题。这种错误很难发现。同样的问题也出现在变量的清除上。因为构造函数与析构函数让我们保证正确地初始化和清除对象（编译器将不允许没有调用构造函数与析构函数就直接创建与销毁一个对象），使我们得到了完全的控制与安全。

集合的初始化同样如此——它防止我们犯那种初始化内部数据类型集合时常犯的错误，使我们的代码更简洁。

编码期间的安全性是C++中的一大问题，初始化和清除是这其中的一个重要部分，随着本书的进展，我们可以看到其他的安全性问题。

4.9 练习

1) 用构造函数与析构函数修改第3章结尾处的HANDLE.H,HANDLE.CPP 和USEHANDL.CPP文件。

2) 创建一个带非缺省构造函数和析构函数的类，这些函数都显示一些信息来表示它们的存在。写一段代码说明构造函数与析构函数何时被调用。

3) 用上题中的类创建一个数组来说明自动计数与集合初始化。在这个类中增加一个显示信息的成员函数。计算数组的大小并逐个访问它们，调用新成员函数。

4) 创建一个没有任何构造函数的类，显示我们可以用缺省的构造函数创建对象。现在为这个类创建一个非缺省的构造函数（带一个参数），试着再编译一次。解释发生的现象。

第5章　函数重载与缺省参数

能使名字方便使用，是任何程序设计语言的一个重要特征。

当我们创建一个对象（即变量）时，要为这个存储区取一个名字。一个函数就是一个操作的名字。正是靠系统描述各种各样的名字，我们才能写出易于人们理解和修改的程序。这在很大程度上就像是写散文——目的是与读者交流。这里就产生了这样一个问题：如何把人类自然语言的有细微差别的概念映射到编程语言中。通常，自然语言中同一个词可以代表许多种不同的含义，这要依赖上下文来确定。这就是所谓的一词多义——该词被重载了。这点非常有用，特别是对于细微的差别。我们可以说“洗衣服，洗汽车”。如果非得说成“洗（洗衣服的洗）衣服，洗（洗汽车的洗）汽车”，那将是很愚蠢的，就好像听话的人对指定的动作毫无辨别能力一样。大多数人类语言都是有冗余的，所以即使漏掉了几个词，我们仍然可以知道话的意思。我们不需要单一的标识——而可以从上下文中理解它的含义。

然而大多数编程语言要求我们为每个函数设定一个唯一的标识符。如果我们想打印三种不同类型的数据：整型、字符型和实型，我们通常不得不用三个不同的函数名，如 print_int()、print_char()和print_float(),这些既增加了我们的编程工作量，也给读者理解程序增加了困难。

在C++中，还有另外一个原因需要对函数名重载：构造函数。因为构造函数的名字预先由类的名字确定，所以只能有一个构造函数名。但如果我们想用几种方法来创建一个对象时该怎么办呢？例如创建一个类，它可以用标准的方法初始化，也可以从文件中读取信息来初始化，我们就需要两个构造函数，一个不带参数（缺省构造函数），另一个带一个字符串作为参数，以表示用于初始化对象的文件的名字。所以函数重载的本质就是允许函数同名。在这种情况下，构造函数是以不同的参数类型被调用的。

重载不仅对构造函数来说是必须的，对其他函数也提供了很大的方便，包括非成员函数。另外，函数重载意味着，我们有两个库，它们都有一个同名的函数，只要它们的参数不同就不会发生冲突。我们将在这一章中详细讨论这些问题。

这一章的主题就是方便地使用函数名。函数重载允许多个函数同名，但还有另一种方法使函数调用更方便。如果我们想以不同的方法调用同一函数，该怎么办呢？当函数有一个长长的参数列表，而大多数参数每次调用都一样时，书写这样的函数调用会使人厌烦，程序可读性也差。C++中有一个很通用的作法叫缺省参数。缺省参数就是在用户调用一个函数时没有指定参数值而由编译器插入参数值的参数。这样 f(“hello”),f(“hi”,1)和f(“howdy”,2,‘c’)可以用来调用同一函数。它们也可能是调用三个已重载的函数，但当参数列表相同时，我们通常希望调用同一函数来完成相同的操作。

函数重载和缺省参数实际上并不复杂。当我们学习完本章的时候，我们就会明白什么时候用到它们，以及编译、连接时它们是怎样实现的。

5.1　范围分解

在第2章中我们介绍了名字范围分解的概念（有时我们用“修饰”这个更通用的术语）。在下面的代码中：

```
void f();
class x {void f();};
```

类x内的函数f（）不会与全局的f（）发生冲突，编译器用不同的内部名 f（）（全局）和x::f()（成员函数）来区分两个函数。在第2章中，我们建议在函数名前加类名的方法来命名函数，所以编译器使用的内部名字可能就是 _f和_x_f。函数名不仅与类名关系密切，而且还跟其他因素有关。

为什么要这样呢？假设我们重载了两个函数名：

```
void print(char);
void print(float);
```

无论这两个函数是某个类的成员函数还是全局函数都无关紧要。如果编译器只使用函数名字的范围，编译器并不能产生单一的内部标识符，这两种情况下都得用 _print结尾。重载函数虽然可以让我们有同名的函数，但这些函数的参数列表应该不一样。所以，为了让重载函数正确工作，编译器要用函数名来区分参数类型名。上面的两个在全局范围定义的函数，可能会产生类似于_print_char和_print_float的内部名。因为，为这样的名字分解规定一个统一的标准毫无意义，所以不同的编译器可能会产生不同的内部名（让编译器产生一个汇编语言代码后我们就可以看到这个内部名是个什么样子了）。当然，如果我们想为特定的编译器和连接器购买编译过的库的话，这就会引起错误。另外，编译器在用不同的方式来产生代码时也可能出现这样的问题。

有关函数重载我们就讲到这里，我们可以对不同的函数用同样的名字，只要函数的参数不同。编译器会通过分解这些名字、范围和参数来产生内部名以供连接器使用。

5.1.1 用返回值重载

读了上面的介绍，我们自然会问：“为什么只能通过范围和参数来重载，为什么不能通过返回值呢？”乍一听，似乎完全可行，同样将返回值分解为内部函数名，然后我们就可以用返回值重载了：

```
void f();
int f();
```

当编译器能从上下文中唯一确定函数的意思时，如int x = f();这当然没有问题。然而，在C中，我们总是可以调用一个函数但忽略它的返回值，在这种情况下，编译器如何知道调用哪个函数呢？更糟的是，读者怎么知道哪个函数会被调用呢？仅仅靠返回值来重载函数实在过于微妙了，所以在C++中禁止这样做。

5.1.2 安全类型连接

对名字的范围分解还可以带来一个额外的好处。这就是，在 C中，如果用户错误地声明了一个函数，或者更糟糕地，一个函数还没声明就调用了，而编译器则按函数被调用的方式去推断函数的声明。这是一个特别严重的问题。有时这种函数是正确的，但如果不正确，就会成为一个很难发现的错误。

在C++中，所有的函数在被使用前都必须事先声明，出现上述情况的机会大大减少了。编译器不会自动为我们添加函数声明，所以我们应该包含一个合适的头文件。然而，假如由于某种原因我们还是错误地声明了一个函数，可能是通过自己手工声明，或包含了一个错误的头文件（也许是一个过期的版本），名称分解会给我们提供一个安全网，也就是人们常说的安全连接。

请看下面的几个例子。在第一个文件中，函数定义是：

```
//: DEF.CPP -- Function definition
  void f(int) {}
```

在第二个文件中，函数没有声明就调用了。

```
//: USE.CPP -- Function misdeclaration
void f(char);
main() {
//! f(1);  //Causes a linker error
}
```

即使我们知道函数实际上应该是f(int),但编译器并不知道，因为它被告知 (通过一个明确的声明) 这个函数是f(char)。因此编译是成功的，在C中，连接也能成功，但在C++中却不行。因为编译器会分解这些名字，这个函数的定义变成了诸如 f_int之类的名字，而使用的函数则是f_char。当连接器试图引用f_char时，它只能找到f_int，所以它就会报告一条出错信息。这就是安全连接。虽然这种问题并不经常出现，但一旦出现就很难发现，尤其是在一个大项目中。这是利用C++编译器查找C语言程序中很隐蔽的错误的一个例子。

5.2 重载的例子

现在我们回过头来看看前面的例子，这里我们用重载函数来改写。如前所述，重载的一个很重要的应用是构造函数。我们可以在下面的stash类中看到这点。

```
//: STASH4.H -- Function overloading
#ifndef STASH4_H_
#define STASH4_H_

class stash {
  int size;  // Size of each space
  int quantity; // Number of storage spaces
  int next; // Next empty space
  // Dynamically allocated array of bytes:
  unsigned char* storage;
  void inflate(int increase);
public:
  stash(int Size); // Zero quantity
  stash(int Size, int InitQuant);
  ~stash();
  int add(void* element);
  void* fetch(int index);
  int count();
};
#endif // STASH4_H_
```

stash()的第一个构造函数与前面一样，但第二个带了一个 Quantity参数来指明分配内存的初始大小。在这个定义中，我们可以看到quantity的内部值与storage指针一起被置零。

```
//: STASH4.CPP -- Function overloading
#include "..\4\stash4.h"
```

```
#include <assert.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

stash::stash(int Size) {
  size = Size;
  quantity = 0;
  next = 0;
  storage = 0;
}

stash::stash(int Size, int InitQuant) {
  size = Size;
  quantity = 0;
  next = 0;
  storage = 0;
  inflate(InitQuant);
}

stash::~stash() {
  if(storage) {
    puts("freeing storage");
    free(storage);
  }
}

int stash::add(void* element) {
  if(next >= quantity) // Enough space left?
    inflate(100); // Add space for 100 elements
  // Copy element into storage,
  // starting at next empty space:
  memcpy(&(storage[next * size]),
         element, size);
  next++;
  return(next - 1); // Index number
}

void* stash::fetch(int index) {
  if(index >= next || index < 0)
    return 0;  // Not out of bounds?
  // Produce pointer to desired element:
  return &(storage[index * size]);
}

int stash::count() {
```

```
  return next; // Number of elements in stash
}

void stash::inflate(int increase) {
  void* v =
    realloc(storage, (quantity+increase)*size);
  assert(v);  // Was it successful?
  storage = (unsigned char*)v;
  quantity += increase;
}
```

当我们用第一个构造函数时，没有内存分配给 storage,内存是在第一次调用 add()来增加一个对象时分配的，另外，执行add()时，当前的内存块不够用时也会分配内存。

下面的测试程序说明了这点，它检查第一个构造函数。

```
//: STSHTST4.CPP -- Function overloading
#include "..\4\stash4.h"
#include <stdio.h>
#include <assert.h>
#define BUFSIZE 80

main() {
  int i;
  FILE* file;
  char buf[BUFSIZE];
  char* cp;
  // ....
  stash intStash(sizeof(int));
  for(i = 0; i < 100; i++)
    intStash.add(&i);
  file = fopen("STSHTST4.CPP", "r");
  assert(file);
  // Holds 80-character strings:
  stash stringStash(sizeof(char) * BUFSIZE);
  while(fgets(buf, BUFSIZE, file))
    stringStash.add(buf);
  fclose(file);

  for(i = 0; i < intStash.count(); i++)
    printf("intStash.fetch(%d) = %d\n", i,
           *(int*)intStash.fetch(i));
  i = 0;
  while(
    (cp = (char*)stringStash.fetch(i++)) != 0)
    printf("stringStash.fetch(%d) = %s",
           i - 1, cp);
  putchar('\n');
}
```

我们可以修改这些代码，增加其他参数来调用第二个构造函数。这样我们可以选择 stash的初始大小。

5.3 缺省参数

比较上面两个stash（）构造函数，它们似乎并没有多大不同，对不对？事实上，第一个构造函数只不过是第二个的一个特例——它的初始大小为零。在这种情况下去创建和管理同一函数的两个不同版本实在是浪费精力。

C++中的缺省参数提供了一个补救的方法。缺省参数是在函数声明时就已给定的一个值，如果我们在调用函数时没有指定这一参数的值，编译器就会自动地插上这个值。在 stash的例子中，我们可以把：

```
stash(int  Size);// zero  quantity
stash(int  Size,int  Quantity);
```

用一个函数声明来代替

```
stash(int Size, int  Quantity=0);
```

这样，stash(int)定义就简化掉了——所需要的是一个单一的stash(int,int)定义。

现在这两个对象的定义

```
stash  A(100),B(100,0);
```

将会产生完全相同的结果。它们将调用同一个构造函数，但对于 A，它的第二个参数是由编译器在看到第一个参数是整型而且没有第二个参数时自动加上去的。编译器能看到缺省参数，所以它知道应该允许这样的调用，就好像它提供第二个参数一样，而这第二个参数值就是我们已经告诉编译器的缺省参数。

缺省参数同函数重载一样，给程序员提供了很多方便，它们都使我们可以在不同的场合使用同一名字。不同之处是，当我们不想亲手提供这些值时，由编译器提供一个缺省参数。上面的那个例子就是用缺省参数代替函数重载的一个很好的例子。用函数重载我们得把一个几乎同样含义、同样操作的函数写两遍甚至更多。当然，如果函数之间的行为差异较大，用缺省参数就不合适了。

在使用缺省参数时必须记住两条规则。第一，只有参数列表的后部参数才可是缺省的，也就是说，我们不可以在一个缺省参数后面又跟一个非缺省的参数。第二，一旦我们开始使用缺省参数，那么这个参数后面的所有参数都必须是缺省的。（这可以从第一条中导出。）

缺省参数只能放在函数声明中，通常在一个头文件中。编译器必须在使用该函数之前知道缺省值。有时人们为了阅读方便在函数定义处放上一些缺省的注释值。如：

```
void  fn(int x  /* =0*/ ) { //...
```

缺省参数可以让声明的参数没有标识符，这看上去很有趣。我们可以这样声明：

```
void  f(int  X, int = 0, float =1.1);
```

在C++中，在函数定义时，我们并不一定需要标识符，像：

```
void f(int X, int,float f)  {/*...*/}
```

在函数体中，x和f可以被引用，但中间的这个参数值则不行,因为它没有名字。这种调用还必须用一个占位符（placeholder）,有f(1)或f(1,2,3.0)。这种语法允许我们把一个参数当作占位符而不去用它。其目的在于我们以后可以修改函数定义而不需要修改所有的函数调用。当然，用一个有名字的参数也能达到同样的目的，但如果我们定义的这个参数在函数体内没有使用，

多数编译器会给出一条警告信息，并认为我们犯了一个逻辑错误。用这种没有名字的参数就可以防止这种警告产生。

更重要的是，如果我们开始用了一个函数参数，而后来发现不需要用它，我们可以高效地将它去掉而不会产生警告错误，而且不需要改动那些调用该函数以前版本的程序代码。

位向量类

这里我们进一步看一个操作符重载和缺省参数的例子。考虑一个高效存储真假标志集合的问题。如果我们有一批数据，这些数据可以用“ on ”或“ off ”来表示。用一个叫位向量的类来存储它们应该是很方便的。有时，位向量并不是作为应用程序的一个工具来使用，而是作为其他类的一部分。

当然对一组标志进行编码，最容易的方法就是每个标志占一个字节，请看下例：

```
//: FLAGS.CPP -- List of true/false flags
#include <stdio.h>
#include <string.h>
#include <assert.h>
#define FSIZE 100
#define TRUE 1
#define FALSE 0

class flags {
  unsigned char f[FSIZE];
public:
  flags();
  void set(int i);
  void clear(int i);
  int read(int i);
  int size();
};

flags::flags() {
  memset(f, FALSE, FSIZE);
}

void flags::set(int i) {
  assert(i >= 0 && i < FSIZE);
  f[i] = TRUE;
}

void flags::clear(int i) {
  assert(i >= 0 && i < FSIZE);
  f[i] = FALSE;
}

int flags::read(int i) {
```

```
  assert(i >= 0 && i < FSIZE);
  return f[i];
}

int flags::size() { return FSIZE; }

main() {
  flags fl;
  for(int i = 0; i < fl.size(); i++)
    if(i % 3 == 0) fl.set(i);
  for(int j = 0; j < fl.size(); j++)
    printf("fl.read(%d)= %d\n", j, fl.read(j));
}
```

然而，这样很浪费存储空间，因为我们用了八位来表示一个只要一位就可表示的标志。有时这种存储很重要，特别是我们想用这个类去建其他的类时。所以下面的 BitVector就用一位表示一个标志。函数重载出现在构造函数和bits()函数中。

```
//: BITVECT.H -- Bit Vector
#ifndef BITVECT_H_
#define BITVECT_H_

class BitVector {
  unsigned char* bytes;
  int Bits, numBytes;
public:
  BitVector(); // Default: 0 size
  // init points to an array of bytes
  // size is measured in bytes
  BitVector(unsigned char* init,
            int size = 8);
  // binary is a string of 1s and 0s
  BitVector(char* binary);
  ~BitVector();
  void set(int bit);
  void clear(int bit);
  int read(int bit);
  int bits(); // Number of bits in the vector
  void bits(int sz); // Set number of bits
  void print(const char* msg = "");
};
#endif // BITVECT_H_
```

第一个构造函数（缺省构造函数）产生了一个大小为零的 BitVector。我们不能在这个向量中设置任何位，因为它们根本就没有位。首先我们必须用重载过的 bits()函数增加这一矢量的大小。这个不带参数的版本将返回向量的当前大小，而 bits(int)会把向量的大小改成参数指定的大小。这样我们可以用同样的函数名来设置和得到向量的大小。注意对新的大小并没有限制

——我们可以增大它，也可以减少它。

第二个构造函数要用到一个无符号字符数组的指针，这也是一个原始字节数组，第二个参数告诉构造函数该数组总共有多少个字节，如果第一个参数是零而不是一个有效的指针，这个数组被初始化为零。如果我们没有给出第二个参数，其缺省值为8。

我们可能以为我们可以用BitVector b(0)这样的声明来产生一个8个字节的BitVector 对象，并把它们初始化为零。如果没有第三个构造函数，情况确实如此。第三个构造函数取 char* 作为它的唯一的参数。参数0既可以用于第二个构造函数（第二个参数缺省）也可用于第三个构造函数。编译器无法知道应该选用哪一个，所以我们会得到一个含义不清的错误。为了正确地产生这样一个BitVector对象，我们必须将零强制转换成一个适当的指针：BitVector b((unsigned char *)0)。这的确有些麻烦，所以我们可以选用BitVector b产生一个空向量，然后把它们扩展到适当的大小，b.bits(64),这就得到8个字节的向量。

编译器必须把char*和unsigned char*当作两个数据类型，这点很重要，否则BitVector(unsigned char*,int)在第二个参数缺省时就和BitVector(char*)一模一样了，编译器无法确定调用哪个函数。

注意print()函数有一个char*型的缺省参数。如果我们知道编译器怎么处理字符串常量，那么这个函数可能让我们感到有点奇怪。编译器在我们每次调用这个函数时都产生一个缺省的字符串吗？答案是否定的。它是在一个特定的保留区内产生一个单一的字符串作为静态全局数据，然后把这个字符串的地址作为缺省值传递给函数的。

一个位串

BitVector的第三个构造函数引用了一个指向字符串的指针，这个字符串代表了一个位串。这样就给用户提供了方便，因为它允许向量的初值可以用自然形式的 0110010来表示。对象产生时会匹配这个串的长度，根据串值来设置或清除每个位。

其他函数还有set()、clear()和read(),这些函数都很重要。它们都用感兴趣的位数作为参数。print()函数打印一条消息，它的缺省参数是空字符串，然后是 BitVector的比特位模型，又一次使用 0和1。

当实现BitVector类时会遇到两个问题。第一个是如果我们需要的位数并不恰好是 8的倍数（或机器的字长），我们必须取最接近的字节数。第二个是在选择某个当前位时要注意。比方，用一个字节数组产生了一个BitVector对象时，数组中的每个字节必须从左读到右，以使我们调用print()函数时出现我们预期的样子。

下面是一组成员函数的定义：

```
//: BITVECT.CPP -- BitVector Implementation
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>
#include "..\4\bitvect.h"
#include <limits.h> //CHAR_BIT = # bits in char
// A byte with the high bit set:
const unsigned char highbit =
  1 << (CHAR_BIT - 1);

BitVector::BitVector() {
  numBytes = 0;
```

```
  Bits = 0;
  bytes = 0;
}
// Notice default args are not duplicated:
BitVector::BitVector(unsigned char* init,
                     int size) {
  numBytes = size;
  Bits = numBytes * CHAR_BIT;
  bytes = (unsigned char*)calloc(numBytes, 1);
  assert(bytes);
  if(init == 0) return; // Default to all 0
  // Translate from bytes into bit sequence:
  for(int index = 0; index<numBytes; index++)
    for(int offset = 0;
        offset < CHAR_BIT; offset++)
      if(init[index] & (highbit >> offset))
        set(index * CHAR_BIT + offset);
}

BitVector::BitVector(char* binary) {
  Bits = strlen(binary);
  numBytes =  Bits / CHAR_BIT;
  // If there's a remainder, add 1 byte:
  if(Bits % CHAR_BIT) numBytes++;
  bytes = (unsigned char*)calloc(numBytes, 1);
  assert(bytes);
  for(int i = 0; i < Bits; i++)
    if(binary[i] == '1') set(i);
}

BitVector::~BitVector() {
  free(bytes);
}

void BitVector::set(int bit) {
  assert(bit >= 0 && bit < Bits);
  int index = bit / CHAR_BIT;
  int offset = bit % CHAR_BIT;
  unsigned char mask = (1 << offset);
  bytes[index] |= mask;
}

int BitVector::read(int bit) {
  assert(bit >= 0 && bit < Bits);
  int index = bit / CHAR_BIT;
  int offset = bit % CHAR_BIT;
```

```
  unsigned char mask = (1 << offset);
  return bytes[index] & mask;
}

void BitVector::clear(int bit) {
  assert(bit >= 0 && bit < Bits);
  int index = bit / CHAR_BIT;
  int offset = bit % CHAR_BIT;
  unsigned char mask = ~(1 << offset);
  bytes[index] &= mask;
}

int BitVector::bits() { return Bits; }

void BitVector::bits(int size) {
  int oldsize = Bits;
  Bits = size;
  numBytes =  Bits / CHAR_BIT;
  // If there's a remainder, add 1 byte:
  if(Bits % CHAR_BIT) numBytes++;
  void* v = realloc(bytes, numBytes);
  assert(v);
  bytes = (unsigned char*)v;
  for(int i = oldsize; i < Bits; i++)
    clear(i); // Erase additional bits
}

void BitVector::print(const char* msg) {
  puts(msg);
  for(int i = 0; i < Bits; i++){
    if(read(i)) putchar('1');
      else putchar('0');
    // Format into byte blocks:
    if((i + 1) % CHAR_BIT == 0) putchar(' ');
  }
  putchar('\n');
}
```

第一个构造函数很简单，就是把所有的变量赋零。第二个构造函数分配内存并初始化位数。接下来用了一个小技巧。外层的 for循环指示字节数组的下标，内层 for循环每次指示这个字节的一位，然而这一位是自左向右用 init[index]&(0x80>>offset)计算出来的。注意这是按位进行与运算的，而且16进制的0x80（最高位为1，其他位为零）右移offset位，产生一个屏蔽码。如果结果不为零，那么在这一位上一定是 1，这个set()函数被用来设置BitVector 内部位。注意描述字节位时应从左到右，只有这样用 print()函数显示的结果看上去才是有意义的。

第三个构造函数把一个二进制0、1序列的字符串转换成一个BitVector。位数就取字符串的

长度。但字符串的长度可能并不正好是8的整数倍，所以字节数numBytes先将位数除以8，然后根据余数是否为0来调整。这种情况下，扫描位是在源串中从左到右进行的，这与第二个构造函数不同。

set()、clear()和read()三个函数形式都差不多，开始三行完全一样：assert()检查传入的参数是否合法，然后产生指向字节数组的索引和指向被选字节的偏移。 Set()和read()用同样的方法产生屏蔽字节：将1移位到所要的位置。但set()是用选定的字节与屏蔽字节相“或”来将该位置1，而read()是用选定的字节与屏蔽字节相“与”来获得该位的状态。clear()是将1移位到指定位来产生屏蔽字节的，然后将选定的字所有的位求“反”(用~)，再与屏蔽字节相“与”，这样只有指定的位被置为零。

注意set()、read()和clear()可以写得更紧凑些，如clear()可以这样来写：

```
bytes[bit/CHAR_BIT]&=~(1<<(bit % CHAR_BIT));
```

这样写可以提高效率，但肯定降低了可读性。

两个重载的bits()函数在行为上差别很大。第一个仅仅是一个存取函数（一种向没有访问权限的人提供私有成员数据的函数），告知数组中共有多少位。第二个函数用它的参数来计算所需的字节数，然后用realloc()函数重新分配内存（如果bytes为零，它将分配新内存），并对新增的位置零。注意，如果我们要求的位数与原有的位数相等，这个函数仍有可能重新分配内存(这取决于realloc()函数的实现)。但这不会破坏任何东西。

print()函数显示msg字串，标准的C库函数puts()已经加了一个新行，所以对缺省参数将输出一个新行。然后它用read()读取每一位的值以确定显示什么字符。为了阅读方便，在每读完8位后它显示一个空格。由于第二个BitVector构造函数是读取字节数组的方式， print()函数将会用熟悉的形式显示结果。

下面的程序通过检验BitVector的所有函数来测试BitVector类。

```
//: BVTEST.CPP -- Testing the BitVector class
#include "..\4\bitvect.h"

main() {
  unsigned char b[] = {
    0x0f, 0xff, 0xf0,
    0xAA, 0x78, 0x11
  };
  BitVector bv1(b, sizeof b / sizeof *b),
    bv2("10010100111100101010001010010010101");
  bv1.print("bv1 before modification");
  for(int i = 36; i < bv1.bits(); i++)
    bv1.clear(i);
  bv1.print("bv1 after modification");
  bv2.print("bv2 before modification");
  for(int j=bv2.bits()-10; j<bv2.bits(); j++)
    bv2.clear(j);
  bv2.set(30);
  bv2.print("bv2 after modification");
  bv2.bits(bv2.bits() / 2);
  bv2.print("bv2 cut in half");
```

```
    bv2.bits(bv2.bits() + 10);
    bv2.print("bv2 grown by 10");
    BitVector bv3((unsigned char*)0);
}
```

对象bv1、bv2、bv3显示了三种不同的BitVector类和它的构造函数。set()和clear()函数也被检验（read()在print()内检验）。在程序的尾部， bv2被减少了一半然后又增大，用以说明将BitVector的尾部置零的一种方法。

我们应该知道在标准的C++库中包含着bits和bitstring类，这些类向位向量提供了一个更完全（也更标准）的实现。

5.4 小结

函数重载和缺省参数都为调用函数提供了方便。有时为弄清到底哪个函数会被调用，也让人迷惑不清。比如在BitVector类中，下式就似乎对两个bits()函数都可以调用：

```
int bits(int sz=-1);
```

如果调用它时不带参数，函数就会用缺省的值 -1，它认为我们想知道当前的位数。这种使用似乎同前面的一样，但事实上存在着明显的不同，至少让我们感觉不舒服。

在bits（）内部我们得按参数的值作一个判断，如果我们必须去找缺省值而不是作为一个普通值，根据这一点，我们就可以形成两个不同的函数。一个是在一般情况下，一个是在缺省情况下。我们也可以把它分割成两个不同的函数体，然后让编译器去选择执行哪一个，这可以提高一点效率，因为不需要传递额外的代码，由条件决定的额外代码也不会被执行。如果我们需要反复调用这个函数，这种效率的少许提高就会表现得很明显。

在这种情况下，用缺省参数我们确实会丢失某些东西。首先，缺省值不能作他用，如本例中-1。现在，我们不能区分一个负数是一个意外还是一个缺省情况。第二，由于在单一参数时只有一个返回值，所以编译器就会丢失很多重载函数时可以得到的有用信息。比如，我们定义：

```
int i=bv1.set(10);
```

编译器会接受它但不再告诉我们其他东西，但作为类的设计者，我们可能认为是一个错误。

再看看用户遇到的问题。当用户读我们的头文件时，哪种设计更容易理解呢？缺省值 -1意味着什么？没有人告诉他们。而用两个分开的函数则非常清楚，因为一个带有一个参数但不返回任何值，而另一个则不带参数但返回一个值。即使没有有关的文档，也很容易猜测这两个函数完成什么功能。

我们不能把缺省参数作为一个标志去决定执行函数的哪一块，这是基本原则。在这种情况下，只要能够，就应该把函数分解成两个或多个重载的函数。缺省参数应该是能把它当作变通值来处理的值，只不过这个值出现的可能比其他值要大，所以用户可以忽略它或只在需要改变缺省值时才去用它。

缺省参数的引用是为了使函数调用更容易，特别是当这些函数的许多参数都有特定值时。它不仅使书写函数调用更容易，而且阅读也更方便，尤其当用户是在制定参数过程中，把那些最不可能调整的缺省参数放在参数表的最后面时。

缺省参数的一个重要应用是在开始定义函数时用了一组参数，而使用了一段时间后发现要增加一些参数。现在我们只要把这些新增参数都作为缺省的参数，就可以保证所有使用这一函

数的代码不会遇到麻烦。

5.5 练习

1) 创建一个message类，其构造函数带有一个 char*型的缺省参数。创建一个私有成员char*，并假定构造函数可以传递一个静态引用串：简单将指针参数赋给内部指针。创建两个重载的成员函数print();一个不带参数，而只是显示存储在对象中的消息，另一个带有 char*参数，它将显示该字符串加上对象内部消息。比较这种方法和使用构造函数的方法，看哪种方法更合理？

2) 测定您的编译器是怎样产生汇编输出代码的，并尝试着减小名字分解表。

3) 用缺省参数修改STASH4.H和STASH4.CPP中的构造函数，创建两个不同的stash对象来测试构造函数。

4) 比较flags类与BitVector类的执行速度。为了保证不会与效率弄混，把 set()、clear()和read()中的index、offset和mask定义合并成一个单一的声明来完成适当的操作（测试这个新的代码以确保代码正确）。

5) 修改FLAGS.CPP以使它可以动态地为标志分配内存，传给构造函数的参数是空间 存储的大小，其缺省值为100。保证在析构函数中清除这些存储空间。

第6章　输入输出流介绍

到目前为止，在这本书里，我们仍使用以前的可靠的C标准I/O库，这是一个可变成类的完美的例子。

事实上，处理一般的I/O问题，比仅仅用标准库并把它变为一个类，需要做更多的工作。如果能使得所有这样的“容器”——标准I/O、文件、甚至存储块——看上去都一样，只须记住一个接口，不是更好吗？这种思想是建立在输入输出流之上的。与标准C输入输出库的各种各样的函数相比，输入输出流更容易、安全、有效。

输入输出流通常是C++初学者学会使用的第一个类库。在这一章里，我们要考察输入输出流的用途，这样，输入输出流要代替这本书剩余部分的C I/O函数。下一章，我们会发现如何建立我们自己的与输入输出流相容的类。

6.1　为什么要用输入输出流

我们可能想知道以前的C库有什么不好。为什么不把C库封装成一个类，然后进行处理？其实，当我们想使C库用起来更安全、更容易一点时，在有些情况下，这样做很完美。例如，当我们想确保一标准输入输出文件总是被安全地打开，被正确地关闭，而不依赖用户是否记得调用close()函数：

```
//: FILECLAS.H -- Stdio files wrapped
#ifndef FILECLAS_H_
#define FILECLAS_H_
#include <stdio.h>

class file {
  FILE* f;
public:
  file(const char* fname, const char* mode="r");
  ~file();
  FILE* fp();
};
#endif // FILECLAS_H_
```

在C中执行文件I/O时，要用一个没有保护的指针指向文件结构。而这个类封装了这个指针，并用构造函数和析构函数保证它能被正确地初始化和清除。第二个构造函数参数是文件模式，其缺省值为“r”，代表“只读”。

在文件I/O函数中，为了取指针的值，可使用fp()访问函数。下面是成员函数的定义：

```
//: FILECLAS.CPP -- Stdio files wrapped
#include <stdlib.h>
#include "..\5\fileclas.h"

file::file(const char* fname, const char* mode){
```

```
  f = fopen(fname, mode);
  if(f == NULL) {
    printf("%s: file not found\n", fname);
    exit(1);
  }
}

file::~file() {
  fclose(f);
}

FILE* file::fp() {
  return f;
}
```

就像常常做的一样，构造函数调用 fopen( )，但它还检查结果，从而确保结果不是零，如果结果是零意味着打开文件时出错。如果出错，这个文件名就被打印，而且函数 exit( ) 被调用。

析构函数关闭这个文件，存取函数 fp( ) 返回值f。下面是使用类文件的一个简单的例子：

```
//: FCTEST.CPP -- Testing class file
#include <assert.h>
#include "..\5\fileclas.h"

main(int argc, char* argv[]) {
  assert(argc == 2);
  file f(argv[1]); // Opens and tests
  #define BSIZE 100
  char buf[BSIZE];
  while(fgets(buf, BSIZE, f.fp()))
    puts(buf);
} // File automatically closed by destructor
```

在该例子中创建了文件对象，在正常C文件I/O函数中通过调用 fp( ) 来使用它。当我们使用完毕时，就忘掉它，则这个文件在该范围的末端由析构函数关闭。

正式的真包装

即使文件指针是私有的，但由于 fp( ) 检索它，所以也不是特别安全的。仅有的保证作用是初始化和销毁。既然这样，为什么不使文件指针是公有的，或者用结构体来代替？注意使用 fp() 得到f的副本时，不能赋值给f —— 它完全处于类控制下。当然，一旦用户使用fp()返回的指针值，他仍能对结构元素赋值。所以安全性是保证一个有效的文件指针而不是结构里的正确内容。

如果想绝对安全，必须禁止用户直接访问文件指针。这意味着所有正常的文件 I/O函数的某些版本将必须作为类成员表现出来。这样，通过C途径所做的每件事，在C++类中均可做到：

```
//: FULLWRAP.H -- Completely hidden file IO
#ifndef FULLWRAP_H_
#define FULLWRAP_H_
```

```
#include <stdio.h>

class File {
  FILE* f;
  FILE* F(); // Produces checked pointer to f
public:
File(); // Create object but don't open file
File(const char* path,
     const char* mode = "r");
~File();
int open(const char* path,
         const char* mode = "r");
int reopen(const char* path,
           const char* mode);
int Getc();
int Ungetc(int c);
int Putc(int c);
int puts(const char* s);
char* gets(char* s, int n);
int printf(const char* format, ...);
size_t read(void* ptr, size_t size,
            size_t n);
size_t write(const void* ptr,
             size_t size, size_t n);
int eof();
int close();
int flush();
int seek(long offset, int whence);
int getpos(fpos_t* pos);
int setpos(const fpos_t* pos);
long tell();
void rewind();
void setbuf(char* buf);
int setvbuf(char* buf, int type, size_t sz);
int error();
void Clearerr();
};
#endif // FULLWRAP_H_
```

这个类包含几乎所有的来自STDIO.H文件的I/O函数。没有vfprintf()函数，它只是被用来实现printf()成员函数。

File有着与前面例子相同的构造函数，而且也有缺省的构造函数。如果要建立一个File对象数组或使用一个File对象作为另一个类的成员，在那个类里，构造函数不发生初始化（但有时在被包含的对象创建后初始化），那么缺省构造函数是重要的。

缺省构造函数设置私有File指针f为0，但是现在，在f的任何引用之前，它的值必须要被检

查以确保它不为0。这是由类的最后一个成员函数F（）来完成的。F（）函数仅由其他成员函数使用，因此是私有的。（我们不想让用户直接访问这个类的File结构）[1]

从任何意义上讲，这不是一个很坏的解决办法。它是相当有效的，可以设想为标准（控制台）I/O和内核格式化（读/写一个存储块而不是一个文件或控制台）构造类似的类。

大的障碍是运行期间用作参数表函数的解释程序。这是在运行期间对格式串做语法分析以及从参数表中索取并解释变量的代码。产生这个问题的四个原因是：

1) 即使仅使用解释程序的一部分功能，所有的东西将获得装载。假如说：

```
printf("%c",'x');
```

我们将得到整个包，包括打印出浮点数和串的那部分。没有办法可减少程序使用的空间。

2) 由于解释发生在运行期间，所以不能终止这个执行。令人灰心的是在编译时，格式串里的所有信息都在这儿，但是直到运行时，才能对其求值。但是，如果能在编译时分析格式串里的变量，就可以建立硬函数调用，它比运行期间解释程序快得多（虽然printf()族函数通常已被优化得很好）。

3) 由于直到运行期间才对格式串求值，一个更糟糕的问题出现了：可能没有编译时的错误检查。如果我们已经尝试找出由于在printf()说明里使用错误的数或变量类型而产生的错误，我们大概对这个问题很熟悉了。C++对编译期间错误检查作了许多工作，使我们及早发现错误，使工作更容易。特别是因为I/O库用得很多，如果弃之不用，那是很不明智的。

4) 对C++，最重要的问题是函数中的printf()族不是能扩展的。它们被设计是用来处理C中四类基本的数据类型（字符，整型，浮点数，双精度及它们的变形）。我们可能认为每增加一个新类时，都能增加一个重载的printf()和scanf()函数（以及它们对文件和串的变量）。但是要记住，重载函数在参数表里必须有不同的类型，printf()族把类型信息隐藏在可变参数表和格式串中。对一个像C++这样其目标是能容易地添加新的数据类型的语言，这是一个笨拙的限制。

6.2 解决输入输出流问题

所有这些问题都清楚地表明：C++中应该有一个最高级别标准类库，用以处理I/O。由于“hello,world”差不多是每个人使用一种新的语言所写的第一个程序，而且由于I/O通常是每个程序的一部分，因此C++中的I/O库必须特别容易使用。这是一个巨大的挑战：它不知道必须适应哪些类，但是它必须能适用于任何新的类。这样的约束要求这个最高级别的类是一个真正的有灵感的设计。

这一章看不到这个设计的细节以及如何向我们自己的类中加入输入输出流功能（在以后的章节里将会看到）。首先，我们必须学会使用输入输出流，其次，在处理I/O和格式时，我们除了能做大量的调节并提高清晰度外，还会看到，一个真正的、功能强大的C++库是如何工作的。

6.2.1 预先了解操作符重载

在使用输入输出流库前，必须明白这个语言的一个新性能，这一性能的详细情况在下一章介绍。要使用输入输出流，必须知道C++中所有操作符具有不同的含义。在这一章，我们特别讲一下“<<”和“>>”，我们说“操作符具有不同的含义”，这值得进一步探讨。

在第5章中，我们已经学到怎样用不同的参数表使用相同的函数名。编译器在编译一个变量后跟一个操作符再后跟一个变量组成的表达式时，它只调用一个函数。那就是说，一个操作

[1] FULLWRAP test file和其实现在此书的源程序中提供，详见前言。

符只不过是不同语法的函数调用。

当然，这就是C++在数据类型方面的特别之处。必须有一个事先说明过的函数来匹配操作符和那些特别变量类型，否则编译器不接受这个表达式。

大多数人发现，操作符重载的麻烦是由于这样的想法：即我们知道 C操作符的所有知识是错的。这是不对的。下面是C++的设计的两个主要目的：

1) 一个用C编译过的程序用C++也能编译。C++编译器仅有的编译错误和警告源于C语言的“漏洞”，修正这些需要局部编辑（其实， C++编译器的提示一般会使我们直接查找到 C程序中未被发现的错误）。

2) 用C++重新编译，C ++编译器不会暗地里改变C程序的行为。

记住这些目的有助于回答许多问题。知道从 C转向C++并不是无规律的变化，会使这种转变更容易。特别是，内部数据类型的操作符将不会以不同的方式工作 —— 不能改变它们的意思。重载的操作符仅能在包含新的数据类型的地方创建。所以能为一个新类建立一个新的重载操作符，但是表达式

```
1<<4；
```

不会突然间改变它的意思，而且非法代码

```
1.414<<1;
```

也不会突然能开始工作了。

6.2.2 插入符与提取符

在输入输出流库里，两个操作符被重载，目的是能容易地使用输入输出流。操作符“ <<”经常作为输入输出流的插入符，操作符“ >>”经常作为提取符。

一个流是一个格式化并保存字节的对象。可以有一个输入流（ istream）或一个输出流(ostream)。有不同类型的输入流和输出流：文件输入流(ifstreams)和文件输出流(ofstreams)、char*内存的（内核格式化）输入流（istrstreams）和输出流(ostrstreams)、以及与标准C++串（string）类接口的串输入流（istringstreams）和串输出流(ostringstreams)。不管在操作文件、标准I/O、存储块还是一串对象中所有这些流对象有相同的接口。单个接口被扩展用于支持新的类。

如果一个流能产生字节（一个输入流），可以用一个提取符从这个流中获取信息。这个提取符产生并格式化目的对象所期望的信息类型。可以使用 cin对象看一下这个例子。 cin对象相当于C中stdin的输入输出流，也就是可重定向的标准输入。不论何时包含了 IOSTREAM.H头文件，这个对象就被预定义了（这样，输入输出流库能自动与大部分编译器连接）。

```
int i;
cin >> i;

float f;
cin >> f;

char c;
cin >> c;

char buf[100];
cin >> buf;
```

每个数据类型都有重载操作符“ >> ”，这些数据类型在一个输入语句里作为“ >> ”右边参数使用（也可以重载我们自己的操作符，这一点将在下一章讲到）。

为了发现各种各样的变量里有什么，我们可以用带插入符“ << ”的cout对象(与标准输出相对应，还有一个与标准错误相对应的cerr对象)：

```
cout << "i = ";
cout << i;
cout << "\n";
cout << "f = ";
cout << f;
cout << "\n";
cout << "c = ";
cout << c;
cout << "\n";
cout << "buf = ";
cout << buf;
cout << "\n";
```

这是特别乏味的，而且对于 printf()函数，看来类型检查没有多大或根本没有改进。幸运的是，输入输出流的重载插入符和提取符，被设计在一起连接成一个更容易书写的复合表达式：

```
cout << "i = " << i << endl;
cout << "f = " << f << endl;
cout << "c = " << c << endl;
cout << "buf = " << buf << endl;
```

在下一章中，我们将明白这是怎样发生的，但是现在，以类用户的态度知道它如何工作也就足够了。

1. 操纵算子

这里已经添加了一个新的元素：一个称作 endl的操纵算子。一个操纵算子作用于流上，这种情况下，插入一新行并清空流（消除所有存储在内部流缓冲区里的还没有输出的字符）。也可以只清空流：

```
cout<<flush；
```

另外有一个基本的操纵算子把基数变为oct(八进制)，dec(十进制)或hex(十六进制)：

```
cout<<hex<<"0x"<<i<<endl；
```

有一个用于提取的操纵算子“ 跳过 ”空格：

```
cin>>ws；
```

还有一个叫 ends的操纵算子和 endl操纵算子一样，仅仅用于 strstreams（稍后介绍）。这些是IOSTREAM.H里所有的操纵算子， IOMANIP.H里会有更多的操纵算子，下一章介绍。

6.2.3 通常用法

虽然cin和提取符“ >> ”为cout和插入符“ << ”提供了一个良好的平衡，但在使用格式化的输入机制，尤其是标准输入时，会遇到与 scanf()中同样的问题。如果输入一个非期望值，进程则被偏离，而且它很难恢复。另外，格式化的输入缺省以空格为分隔符。这样，如果把上面

的代码块搜集成一个程序：

```
//: IOSEXAMP.CPP -- Iostream examples
#include <iostream.h>

main() {
  int i;
  cin >> i;

  float f;
  cin >> f;

  char c;
  cin >> c;

  char buf[100];
  cin >> buf;

  cout << "i = " << i << endl;
  cout << "f = " << f << endl;
  cout << "c = " << c << endl;
  cout << "buf = " << buf << endl;

  cout << flush;
  cout << hex << "0x" << i << endl;
}
```

给出以下输入：

```
12 1.4 c this is a test
```

我们应该得到与输入相同的输出

```
12
1.4
c
this is a test
```

但输出是：（有点不是所期望的）

```
i = 12
f = 1.4
c = c
buf = this
0xc
```

注意到buf只得到第一个字，这是由于输入机制是通过寻找空格来分隔输入的，而空格在“this”的后面。另外，如果连续的输入串长于为buf分配的存储空间，会发生buf溢出现象。

看来提供cin和提取符只是为了完整性，这可能是查看它的一个好方法。实际上，有时想得到列字符的一行一行排列的输入，然后扫描它们，一旦它们安全地处于缓冲区就执行转换。这样，我们就不必担心输入机制因非期望数据而阻塞了。

另一件要考虑的事是命令行接口的整体概念。当控制台还比不上一台玻璃打字机时人们就对这一点有所认识。世界发展迅速，现在图形用户接口（GUI）处于支配地位。这样的世界里，控制台I/O的意思是什么呢？除了很简单的例子或测试外，可以完全忽略 cin，并采取以下更有意义的步骤：

1) 如果程序需要输入，从一个文件中读取输入——会发现使用输入输出流文件非常容易。输入输出流在GUI（图形用户接口）下仍运行得很好。

2) 只读输入而不想去转换它。一旦在某处获得输入，它在转换时不会影响其他，那么我们可以安全地扫描它。

3) 输出的情况有所不同。如果正在使用图形用户接口，则必须将输出送到一个文件中（这与将输出送到cout是相同的）或者使用图形用户接口设备显示数据，cout不工作。然而，通常把输出送到cout是有意义的。在这两种情况下，输入输出流的输出格式化函数都是很有用的。

6.2.4 面向行的输入

要获取一行输入，有两种选择：成员函数 get（）或getline（）。两个函数都有三个参数：指向存储结果字符的缓冲区指针、缓冲区大小（不能超过其限度）和知道什么时候停止读输入的终止符。终止符有一个经常用到的缺省值“ \n ”。两个函数遇到输入终止符时，都把零储存在结果缓冲区里。

其不同点是什么呢？差别虽小但极其重要： get（）遇到输入流的分隔符时就停止，而不从输入流中提取分隔符。如果用同样的分隔符再调用一次 get（）函数，它会立即返回而不带任何输入。（要么在下一个get（）说明里用一个不同的分隔符，要么用一个不同的输入函数）。getline（）与其相反，它从输入流中提取分隔符，但仍没有把它储存在结果缓冲区里。

总之，当我们在处理文本文件时，无论什么时候读出一行，都会想到用 getline（）。

1. get（）的重载版本

get（）有三种其他的重载版本：一个没有参数表，返回下一个字符，用的是一个 int返回值；一个把字符填进字符参数表，用一个引用（想立即弄明白这个，要跳到第 10章看）；一个直接存储在另一个输入输出流对象的基本缓冲区结构里。这一点在本章的后面介绍。

2. 读原始字节

如果想确切知道正在处理什么，并把字节直接移进内存中的变量、数组或结构中，可以用read（）函数。第一个参数是一个指向内存目的地址的指针，第二个参数指明要读的字节数。预先将信息存储在一个文件里特别有用。例如，在二进制形式里，对一个输出流使用相反的write（）成员函数。以后我们会看到所有这些函数的例子。

3. 出错处理

除没有参数表的 get（）外。所有 get（）和getline（）的版本都返回字符来源的输入流，没有参数表的get（）返回下一个字符或EOF。如果取回输入流对象，要询问一下它是否正确。事实上，我们可用成员函数 good（）、eof（）、fail（）和bad（）询问任何输入输出流是否正确。这些返回状态信息基于 eofbit（指缓冲位于序列的末尾）、failbit（指由于格式化问题或不影响缓冲区的其他问题而使操作失败）和badbit（指缓冲区出错）。

然而，正如前面提到的，如果想输入特定类型，而且从输入中读出的类型与所期望的不一致时，输入流的状态一般要遭到莫名其妙的破坏。当然可通过处理输入流来改正这个问题。如果大家按照我的建议，一次读入一行或一个大的字符段（用 read（）函数），并且除简单情况外不使用输入格式函数，那么，所关心的只是读入位置是否处于输入的末尾。幸好这种测试很

简单，而且能在条件内部完成，如while(cin)和if(cin)等。要接受这样的事实，当我们在上下文环境中使用输入流对象时，正确值被安全、正确和魔术般地产生出来，以表明对象是否达到输入的末尾。我们也可以像if（！cin）一样，使用布尔非运算“！”，表明这个流不正确，即我们可能已经达到输入的末尾了，应该停止读这个流。

有时候，流处于非正确状态，我们知道这个情况，并且想继续使用它。例如，如果我们到达文件的末尾，eofbit和failbit被设置，这样，那个流对象的情况表明：这个流不再是正确的了。我们可能想通过寻找以前的位置并读出更多的数据而继续使用这个文件。要改变这个情况，只需简单地调用clear()成员函数即可[1]。

6.3 文件输入输出流

用输入输出流操作文件比在C中用STDIO.H要容易得多，安全得多。要打开文件，所做的就是建立一个对象。构造函数做打开文件的工作。不必明确地关闭一个文件（尽管我们用close()成员函数也能做到）。这是因为当对象超出范围时，析构函数将其关闭。要建立一缺省输出文件，只要建一个ofstream对象即可。下面这个例子说明到目前为止已讨论的很多特性。注意FSTREAM.H文件包含声明文件I/O类，这也包含IOSTREAM.H文件。

```
//: STRFILE.CPP -- Stream I/O with files
// The difference between get() & getline()
#include <fstream.h>  // Includes iostream.h
#include <assert.h>
#define SZ 100  // Buffer size

main() {
  char buf[SZ];
  {
    ifstream in("strfile.cpp"); // Read
    assert(in); // Ensure successful open
    ofstream out("strfile.out"); // Write
    assert(out);
    int i = 1; // Line counter

    // A less-convenient approach for line input:
    while(in.get(buf, SZ)) { // Leaves \n in input
      in.get(); // Throw away next character (\n)
      cout << buf << endl; // Must add \n
      // File output just like standard I/O:
      out << i++ << ": " << buf << endl;
    }
  } // Destructors close in & out

  ifstream in("strfile.out");
  assert(in);
```

[1] 更新的实现将支持输入输出流的这种处理错误的方式，但某些情况下也会弹出异常。

```
// More convenient line input:
while(in.getline(buf, SZ)) { // Removes \n
  char* cp = buf;
  while(*cp != ':')
    cp++;
  cp += 2; // Past ": "
  cout << cp << endl; // Must still add \n
 }
}
```

建立ifstream和ofstream，后跟一个assert()以保证文件能成功地被打开。还有一个对象，用在编译器期望一个整型结果的地方，产生一个表明成功或失败的值。 (这样做要调用一个自动类型转换成员函数,这一点将在第11章讨论)。

第一个while循环表明get()函数的两种形式的用法。一是不论读到第 SZ-1个字符还是遇到第三个参数（缺省值为“ \n ”），get（）函数把字符取进一个缓冲区内，并放入一个零终止符。get（）把终止符留在输入流内，这样，通过使用不带参数形式的 get（），这个终止符必然通过in.get()而被扔掉，这个 get（）函数取回一个字节并使它作为一个 int类型返回。二是可以用ignore（）成员函数，它有两个缺省的参数，第一个参数是扔掉字符的数目，缺省值是 1，第二参数表示ignore()函数退出处的那个字符（在提取后），缺省值是EOF。

下面将看到两个看起来很类似的输出说明： cout和out。注意这是很合适的，我们不必担心正在处理的是哪种对象，因为格式说明对所有的 ostream对象同样有效。第一类输入回显行至标准输出，第二类写行至新文件并包括一个行数目。

为说明getline（），打开我们刚刚建立的文件并除去行数是一件很有趣的事。在打开一个要读的文件之前，为确保文件是正当关闭的，有两种选择。可以用大括号包住程序的第一部分以迫使out对象脱离范围，这样，调用析构函数并在这里关闭这个文件。也可以为两个文件调用close（），如想这样做，可以调用open（）成员函数重用in对象（也可以像第12章讲的那样，在堆里动态地创建和消除对象）。

第二个while循环说明getline（）如何从其遇到的输入流中移走终止符（它的第三个参数缺省值是“ \n ”）。然而getline()像get()一样，把零放进缓冲区，而且它同样不能插入终止符。

打开方式

可以通过改变缺省变量来控制文件打开方式。下表列出控制文件打开方式的标志。

标　志	函　数
ios::in	打开一个输入文件，用这个标志作为 ifstream的打开方式，以防止截断一个现成的文件
ios::out	打开一个输出文件，当用于一个没有 ios::app、ios::ate或ios::in的ofstream时，ios::trunc被隐含
ios::app	以追加的方式打开一个输出文件
ios::ate	打开一现成文件（不论是输入还是输出）并寻找末尾
ios::nocreate	仅打开一个存在的文件（否则失败）
ios::noreplace	仅打开一个不存在的文件（否则失败）
ios::trunc	如果一个文件存在，打开它并删除旧的文件
ios::binary	打开一个二进制文件，缺省的是文本文件

这些标志可用一个“位或”（OR）运算来连接。

6.4 输入输出流缓冲

无论什么时候建立一个新类，都应当尽可能努力地对类的用户隐藏类的基本实现详情，仅显示他们需要知道的东西，把其余的变为私有以避免产生混淆。通常，当我们使用输入输出流的时候，我们不知道或不关心在哪个字节被产生或消耗。其实，无论正在处理的是标准 I/O、文件、内存还是某些新产生的类或设备，情况都有所不同。

这时，重要的是把消息发送到产生和消耗字节的输入输出流。为了提供通用接口给这些流并且仍然隐藏其基本的实现，它被抽像成自己的类，叫streambuf。每一个输入输出流都包含一个指针，指向某种streambuf（这依赖于它是否处理标准I/O、文件、内存等等）。我们可以直接访问streambuf。例如，可以向streambuf移进、移出原始字节，而不必通过输入输出流来格式化它们。当然，这是通过调用streambuf对象的成员函数来完成的。

当前，我们要知道的最重要的事是：每个输入输出流对象包含一个指向 streambuf的指针，而且，如果需要调用的话，streambuf有我们可以调用的成员函数。

为了允许我们访问streambuf，每个流对象有一个叫做rdbuf()的成员函数，这个函数返回指向对象的streambuf的指针。这样，我们可以为下层的 streambuf调用任何成员函数。然而，对streambuf指针所做的最有兴趣的事之一是：使用“ << ”操作符将其与另一个输入输出流联结。这使我们的对象中的所有字节流进“ << ”左边的对象中。这意味着，如果把一个输入输出流的所有字节移到另一个输入输出流，我们不必做读入它们的一个字节或一行这样单调的工作。这是一流的方法。

例如，下面是打开一个文件并将其内容发送到标准输出（类似前面的例子）的一个很简单的程序：

```
//: STYPE.CPP -- Type a file to standard output
#include <fstream.h>
#include <assert.h>

main(int argc, char* argv[]) {
  assert(argc == 2); // Must have a command line
  ifstream in(argv[1]);
  assert(in); // Exits if it doesn't exist
  cout << in.rdbuf(); // Outputs entire file
}
```

在确信命令行有一个参数后，通过使用这个变数建立一个文件输入流 ifstream。如果这个文件不存在，打开它时将会失败，这个失败被assert(in)捕获。

所有的工作实际上在这个说明里完成：

```
cout<<in.rdbuf()；
```

它把文件的整个内容送到cout。这不仅比代码更简明扼要，也比在每次移动字节更加有效。

使用带streambuf的get()函数

有一种get（）形式允许直接向另一对象的 streambuf写入。第一个参数是streambuf的目的

地址（它的地址神秘地由一个引用携带，第 10章讨论这个问题）。第二个参数是终止符，它终止get（）函数。所以，打印一个文件到标准输出的另一方法是：

```
//: SBUFGET.CPP -- Get directly into a streambuf
#include <fstream.h>

main() {
  ifstream in("sbufget.cpp");
  while(in.get(*cout.rdbuf()))
    in.ignore();
}
```

rdbuf()返回一个指针，它必须逆向引用，以满足这个函数看到对象的需要。 get（）函数不从输入流中拖出终止符，必须通过调用 ignore（）移走终止符。所以， get()永远不会跳到新行上去。

我们可能不必经常使用这样的技术，但知道它的存在是有用的。

6.5 在输入输出流中查找

每种输入输出流都有一个概念：“ 下一个 ”字符来自哪里（若是输入流）或去哪里（若是输出流）。在某些情况下,可能需要移动这个流的位置，可以用两种方式处理：第一种方式是在流里绝对定位，叫流定位（streampos）；第二种方式像标准C库函数fseek()那样做，从文件的开始、结尾或当前位置移动给定数目的字节。

流定位（streampos）方法要求先调用“ tell ”函数：对一个输出流用 tellp（）函数，对一个输入流用tellg（）函数。(“ p ”指“ 放指针 ”, “ g ”指“ 取指针 ”)。要返回到流中的那个位置时，这个函数返回一个streampos，我们以后可以在用于输出流的 seekp（）函数或用于输入流的seekg（）函数的单参数版本里使用这个streampos。

另一个方法是相对查找，使用 seekp（）和seekg（）的重载版本。第一个参数是要移动的字节数，它可以是正的或负的。第二个参数是查找方向：

Ios::beg	从流的开始位置
Ios::cur	从流的当前位置
Ios::end	从流的末尾位置

下面是一个说明在文件中移动的例子，记住，不仅限于在文件里查找，就像在 C和C++中的STDIO.H一样。在C++中，我们可以在任何类型的流中查找(虽然查找时， cin和cout的方式未被定义)：

```
//: SEEKING.CPP -- Seeking in iostreams
#include <fstream.h>
#include <assert.h>

main(int argc, char* argv[]) {
  assert(argc == 2);
  ifstream in(argv[1]);
  assert(in); // File must already exist
  in.seekg(0, ios::end); // End of file
  streampos sp = in.tellg(); // Size of file
```

```
  cout << "file size = " << sp << endl;
  in.seekg(-sp/10, ios::end);
  streampos sp2 = in.tellg();
  in.seekg(0, ios::beg); // Start of file
  cout << in.rdbuf(); // Print whole file
    in.seekg(sp2); // Move to streampos
    // Prints the last 1/10th of the file:
    cout << endl << endl << in.rdbuf() << endl;
}
```

这个程序从命令行中读取文件名，并作为一个文件输入流（ifstream）打开。assert（）检测打开是否失败。由于这是一种输入流，因此用 seekg（）来定位“取指针”，第一次调用从文件末查找零字节，即到末端。由于 streampos是一个long的typedef，那里调用tellg（），返回被打印文件的大小。然后执行查找，移取指针至文件大小的 1/10处——注意那是文件尾的反向查找，所以指针从尾部退回。如我们想进行从文件尾的正向查找，取指针刚好停在文件尾。那里的streampos被读进sp2，然后，seekg（）会到文件开始处执行，整个过程可通过由rdbuf（）产生的streambuf指针打印出来。最后，seekg（）的重载版与streampos sp2一起使用，移到先前的位置，文件的最后部分被打印出来。

建立读/写文件

既然了解streambuf并知道怎样查找，我们会明白怎样建立一个既能读又能写文件的流对象。下面的代码首先建立一个有标志的 ifstream，它既是一个输入文件又是一个输出文件，编译器不会让我们向ifstream写，因此，需要建立具有基本流缓冲区的输出流（ostream）：

```
ifstream in("filename", ios::in|ios::out);
ostream out(in.rdbuf());
```

我们可能想知道向这样一个对象写内容时，会发生什么。下面是一个例子：

```
//: IOFILE.CPP -- Reading & writing one file
#include <fstream.h>
main() {
  ifstream in("iofile.cpp");
  ofstream out("iofile.out");
  out << in.rdbuf(); // Copy file
  in.close();
  out.close();
  // Open for reading and writing:
  ifstream in2("iofile.out",ios::in|ios::out);
  ostream out2(in2.rdbuf());
  cout << in2.rdbuf();  // Print whole file
  out2 << "Where does this end up?";
  out2.seekp(0, ios::beg);
  out2 << "And what about this?";
  in2.seekg(0, ios::beg);
  cout << in2.rdbuf();
}
```

前五行把这个程序的源代码拷贝进一个名叫 iofile.out的文件，然后关闭这个文件。这给了我们一个可在其周围操作的安全的文本文件。那么前面提及的技术被用来建立两个对象，这两个对象向同一个文件读和写。在 cout<<in2.rdbuf()里，可看到“ 取 ”指针在文件的开始被初始化。“ 放 ”指针被放到文件的末尾，这是由于“ Where does this end up? ”追加到这个文件里。然而，如果“ 放 ”指针移到 seekp（）的开始处，所有插入的文本覆盖现成的文本。当“ 取 ”指针用seekg（）移回到开始处时，两次写结果均可见到，而且文件被打印出来。当然，当 out2 脱离范围时，析构函数被调用，这个文件被自动保存和关闭。

6.6 strstreams

第三个标准型输入输出流可直接与内存而不是一个文件或标准输出一起工作。它允许我们用同样的读函数和格式函数去操作内存里的字节。旧式的计算机，内存是指内核，所以，这种功能有时叫内核格式。

strstream的类名字回显文件流的类名字。如想建立一个从中提取字符的 strstream，我们就建立一个istrstream。如想把字符放进一个strstream，我们就建立一个ostrstream。

串流与内存一起工作，所以我们必须处理这样的问题：内存来自哪里又去哪里。这个问题并不复杂到令人害怕的程度，但我们必须弄懂并注意它。从 strstreams中得到的好处远大于这一微小的不利。

6.6.1 为用户分配的存储

由用户负责分配存储空间，恰好是弄懂这个问题的最容易的途径。用 istrstreams，这是唯一允许的方法。下面是两个构造函数：

```
istrstream::istrstream(char* buf);
istrstream::istrstream(char* buf, int size);
```

第一个构造函数取一个指向零终止符数组的指针；我们可以提取字节直到零为止。第二个构造函数另外还需要这个数组的大小，这个数组不必是零终止的。我们可以一直提取字节到buf[size]，而不管是否遇到一个零。

当移交数组地址给一个istrstream构造函数时，这个数组必须已经填充了我们要提取的并且假定格式化成某种其他数据类型的字符。下面是一个简单的例子 [1]：

```
//: ISTRING.CPP -- Input strstreams
#include <strstrea.h>

main() {
  istrstream s("1.414 47 This is a test");
  int i;
  float f;
  s >> i >> f; // Whitespace-delimited input
  char buf2[100];
  s >> buf2;
  cout << "i = " << i << ", f = " << f;
```

[1] 注意文件名必须被截断，以处理DOS对文件名的限制。如果我们的系统支持长文件名，我们必须调整头文件名（否则只拷贝头文件）。

```
  cout << " buf2 = " << buf2 << endl;
  cout << s.rdbuf(); // Get the rest...
}
```

比起标准C库里atof()、atoi()等等这样的函数，可以看到，这是把字符串转换成类型值更加灵活和更加一般的途径。

编译器在下面的代码里处理这个串的静态存储分配：

```
istrstream s("1.414 47 This is a test");
```

我们还可以移交一个在栈或堆里分配的有零终止符的指针给它。

在s>>i>>f里，第一个数被提取到i，第二数被提取到f，这不是“字符的第一个空白分隔符”，这是因为它依赖于它正被提取的数据类型。例如，如果这个串被替换成“1.414 47 This is a test”，那么i取值1，这是因为输入程序停留在小数点上。然后 f取0.414。把一浮点数分成整数部分和小数部分，是有用的。否则看起来似乎是一个错误。

就像已猜测的一样，buf2没有取串的其余部分，仅取空白分隔符的下一个字。一般地，使用输入输出流提取符最好是当我们仅知道输入流里的确切的数据序列并且正在转换某些类型而不是一字符串。然而，如果我们想立即提取这个串的其余部分并把它送给另一个输入输出流，我们可以使用显示过的rdbuf（）。

输出strstreams也允许我们提供自己的存储空间。在这种情况下，字节在内存中被格式化。相应的构造函数是：

```
ostrstream::ostrstream(char*,int,int=ios::out);
```

第一个参数是预分配的缓冲区，在那里字符将结束，第二个参数是缓冲区的大小，第三个参数是模式。如果模式是缺省值，字符从缓冲区的开始地址格式化。如果模式是 ios::ate或ios::app（效果一样），字符缓冲区被假定已经包含了一个零终止字符串，而任何新的字符只能从零终止符开始添加。

第二个构造函数参数表示数组大小，且被对象用来保证它不覆盖数组尾。如我们已填满数组而又想添加更多的字节，这些字节是加不进去的。

关于ostrstreams，记住重要的是：没有为我们插入一般在字符数组末尾所需要的零终止符。当我们准备好零终止符时，用特别操纵算子ends。

一旦已建立一个ostrstream，就可以插入我们需要插入的任何东西，而且它将在内存缓冲区里完成格式化。下面是一个例子：

```
//: OSTRING.CPP -- Output strstreams
#include <strstrea.h>
#define SZ 100

main() {
cout << "type an int, a float and a string:";
int i;
float f;
cin >> i >> f;
cin >> ws; // Throw away white space
char buf[SZ];
cin.getline(buf, SZ); // Get rest of the line
```

```
  // (cin.rdbuf() would be awkward)
  ostrstream os(buf, SZ, ios::app);
  os << endl;
  os << "integer = " << i << endl;
  os << "float = " << f << endl;
  os << ends;
  cout << buf;
  cout << os.rdbuf(); // Same effect
  cout << os.rdbuf(); // NOT the same effect
}
```

这类似于前面int和float的例子。我们可能认为取一行其余部分的逻辑方法是使用 rdbuf（）；这个当然可以，但它很笨拙，因为所有包括回车的输入一直被收集起来，直到用户按control-Z(在unix中control-D)表明输入结束时才停下来。使用getline（）所表明的方法，一直取输入直到用户按下回车才停下来。这个输入被取进 buf，buf用来构造ostrstream os 。如果未提供第三个参数ios::app，构造函数缺省地写在buf的开头，覆盖刚被收集的行。然而，“追加”标志使它把被格式化后的信息放在这个串的末尾。

像其他的输出流一样，可以用平常的格式化工具发送字节到 ostrstream。区别是仅用ends在末尾插入零。注意，endl是在流中插入一个新行，而不是插入零。

现在信息在buf里格式化，可用cout<<buf直接发送它。然而，也有可能用os.rdbuf()发送它。当我们这样做的时候，在streambuf里的“取”指针随这字符被输出而向前移动。正因如此，第二次用到cout<<os.rdbuf()时，什么也没有发生——“取”指针已经在末端。

6.6.2 自动存储分配

输出strstreams（但不是istrstreams）是另一种分配存储空间的方法：它们自己完成这个操作，我们所做的是建立一个不带构造函数参数的ostrstream：

```
ostrstream A;
```

现在，A关心它自己在堆中存储空间的分配，可以在 A中想放多少字节就放多少字节，它用完存储空间，如有必要，它将移动存储块以分配更多的存储空间。

如果不知道需要多少空间，这是一个很好的解决办法，因为它很灵活。格式化数据到strstream，然后把它的streambuf移给另一个输入输出流。这样做很完美：

```
A << "hello, world. i = " << i << endl << ends;
cout << A.rdbuf();
```

这是所有解决办法中最好的。但是，如果要 A的字符被格式化到内存物理地址，会发生什么呢？这是很容易做到的——只要调用 str（）成员函数即可：

```
char* cp=A.str();
```

还有一个问题，如果想在 A中放进更多的字符会发生什么呢？如果我们知道 A分配的存储空间足够放进更多的字符，就好了，但那是不正确的。一般地，如给 A更多的字符，它将用完存储空间。通常 A试图在堆中分配更多的存储空间，这经常需要移动存储块。但是流对象刚刚把它的存储块的地址交给我们，所以我们不能很好地移动那个块，因为我们期望它处于特定的位置。

ostrstream处理这个问题的方法是“冻结”它自己。只要不用 str（）请求内部char*，就可

以尽可能向串输出流中追加字符。它将从堆中分配所需的存储空间，当对象脱离作用域时，堆存储空间自动被释放。

然而，如果调用str（），ostrstream就“冻结”了，就不能再向给它添加字符，无需通过实现来检测错误。向冻结的ostrstream添加字符导致未被定义的行为。另外，ostrstream不再负责清理存储器。当我们用str()请求char*时，要负责清除存储器。

为了不让内存泄漏，必须清理存储器。有两种清理办法。较普通的办法是直接释放要处理的内存。为搞懂这个问题，我们得预习一下C++中两个新的关键字：new和delete。就像第12章学到的一样，这两个关键字用得相当多，但目前可认为它们是用来替代C中malloc()和free()的。操作符new返回一个存储块，而delete释放它。重要的是必须知道它们，因为实际上C++中所有内存分配是由new完成的。ostrstream也是这样的。如果内存是由new分配的，它必须由delete释放。所以，如果有一个ostrstream A，用str()取得char*，清理存储器的办法是：

```
delete A.str();
```

这能满足大部分需要，但还有另一个不是很普通的释放存储器的办法：解冻ostrstream，可通过调用freeze()来做。freeze()是ostrstream的streambuf成员函数。freeze有一个缺省参数，这个缺省参数冻结这个流。用零参数对它解冻：

```
A.rdbuf()->freeze(0);
```

当A脱离作用域时，存储被重新分配，而且它的析构函数被调用。另外，可添加更多的字节给A。但是这可能引起存储移动，所以最好不要用以前通过调用str（）得到的指针——在添加更多的字符后，这个指针将不可靠。

下面的例子测试在一个流被解冻后追加字符的能力：

```
//: WALRUS.CPP -- Freezing a strstream
#include <strstrea.h>

main() {
  ostrstream s;
  s << "'The time has come', the walrus said,";
  s << ends;
  cout << s.str() << endl; // String is frozen
  // S is frozen; destructor won't delete
  // the streambuf storage on the heap
  s.seekp(-1, ios::cur); // Back up before NULL
  s.rdbuf()->freeze(0); // Unfreeze it
  // Now destructor releases memory, and
  // you can add more characters (but you
  // better not use the previous str() value)
  s << " 'To speak of many things'" << ends;
  cout << s.rdbuf();
}
```

在放第一个串到s后，添加一个ends，所以这个串能用由str()产生的char*打印出来。在这个意义上，s被冻结了。为了添更多的字节给s，“放”指针必须后移一步，这样下一个字符被放到由ends插入的零的上面。（否则，这个串仅被打印到原来的零的上面）。这是由seekp（）完成的。然后通过使用rdbuf()和调用freeze(0)取回基本streambuf指针，s被解冻。在这个意义上，

s就像它以前调用str()一样：我们可以添加更多的字符，清理由析构函数自动完成。

解冻一个ostrstream并继续添加字符是有可能的，但通常不这样做。正常的情况下，如果我们获得ostrstream的char*时想添加更多的字符，就建立一个新的ostrstream，通过使用rdbuf()把旧的流灌到这个新的流中去，并继续添加新的字符到新的ostrstream中。

1. 检验移动

如果我们仍不相信调用str()就得对ostrstream的存储空间负责，下面的例子说明存储定位被移动了，因而由str（）返回的旧指针是无效的：

```
//: STRMOVE.CPP -- Ostrstream memory movement
#include <strstrea.h>

main() {
  ostrstream s;
  s << "hi";
  char* old = s.str(); // Freezes s
  s.rdbuf()->freeze(0); // Unfreeze
  for(int i = 0; i < 100; i++)
    s << "howdy"; // Should force reallocation
  cout << "old = " << (void*)old << endl;
  cout << "new = " << (void*)s.str(); // Freezes
  delete s.str(); // Release storage
}
```

在插入一个串到s中并用str（）捕获char*后，这个串被解冻而且有足够的新字节被插入，真正确保了内存被重新分配且大多数被移动。在打印出旧的和新的 char*值后，存储明确地由delete释放，因为第二次调用str()又冻结了这个串。

为了打印出它们指向的串的地址而不是这个串，必须把 char*指派为void*。char*的操作符“ << ”打印出它正指向的串，而对应于void*的操作符“ << ”打印出指针的十六进制表示值。

有趣的是应注意到：在调用 str()前，如不插一个串到s中，结果则为0。这意味着直到第一次插入字节到ostrstream时，存储才被重新分配。

2. 一个更好的方法

标准C++ string类以及与其联系在一起工作的stringstream类，对解决这个问题做了很大的改进。使用这两个类代替char*和strstream时，不必担心负责存储空间的事——一切都被自动清理[1]。

6.7 输出流格式化

全部的努力以及所有这些不同类型的输入输出流的目的，是让我们容易地从一个地方到另一个地方移动并翻译字节。如果不能用 printf()族函数完成所有的格式化，这当然是没有用的。我们将学到输入输出流所有可用的输出格式化函数，得到所需要的那些字节。

输入输出流的格式化函数开始有点使人混淆，这是因为经常有一种以上的方式控制格式化：通过成员函数控制，也可以通过操纵算子来控制。更容易混淆的是，有一个派生的成员函数设置控制格式化的状态标志，如左对齐或右对齐，对十六进制表示法是否用大写字母，是否总是用十进制数表示浮点数的值等等。另一方面，这里有特别的成员函数用以设置填充字符、域宽

[1] 这本书中，这些类仅是草稿，不能在编译器上实现。

和精度，并读出它们的值。

6.7.1 内部格式化数据

ios类（在头文件IOSTREAM.H中可看到）包含数据成员以存储属于那个流的所有格式化数据。有些数据的值有一定范围并被储存在变量里：浮点精度、输出域宽度和用来填充输出（通常是一空格）的字符。格式化的其余部分是由标志所决定的，这些标志通常被连在一起以节省空间，并一起被指定为格式标志。可以用 ios::flags()成员函数发现格式化标志的值，这个成员函数没带参数并返回一个包含当前格式化标志的 long(typedefed to fmtflags)型值。函数的所有其余部分使格式化标志发生改变并返回格式化标志先前的值。

```
fmtflags ios::flags(fmtflags newflags);
fmtflags ios::setf(fmtflags ored_flag);
fmtflags ios::unsetf(fmtflags clear_flag);
fmtflags ios::setf(fmtflags bits, fmtflags field);
```

有时第一个函数迫使所有的标志改变。更多的是每次用剩下的三个函数来改变一个标志。

setf()的用法看来更加令人混淆：要想知道用哪个重载版本，必须知道正要改变的是哪类标志。这里有两类标志：一类是简单的 on或off，一类是与其他标志在一个组里工作的标志。on/off标志理解起来最简单，因为我们可用 setf(fmtflags)将它们变为on，用unsetf(fmtflags)将它们变为off。这些标志是：

on/off标志	作　　用
ios::skipws	跳过空白字符（对于输入这是缺省值）
ios::showbase	打印一个整数值时标明数值基数（十进制，八进制或十六进制），使用的格式能被C++编译器读出
ios::showpoint	表明浮点数的小数点和后面的零
ios::uppercase	显示十六进制数值的大写字母 A-F和科学记数法中的大写字母 E
ios::showpos	显示加号（+）代表正值
ios::unitbuf	“设备缓冲区”；在每次插入操作后，这个流被刷新
ios::stdio	使这个流与C标准I/O系统同步

例如，为 cout显示加号，可写成 cout.setf(ios::showpos)；停止显示加号，可写成 cout.unsetf(ios::showpos)。应该解释一下最后两个标志。当一个字符一旦被插进一个输出流，如果想确信它是一个输出时，可启用缓冲设备。也可以不用缓冲输出，但用缓冲设备会更好。

有一个程序用了输入输出流和C标准I/O库（用C库不是不可能的），标志ios::stdio就被采用。如果发现输入输出流的输出和printf()输出出现了错误的次序，就要设置这个标志。

1. 格式域

第二类格式化标志在一个组里工作，一次只能用这些标志中的一种，就像旧式的汽车收音机按钮一样 —— 按下一个按钮，其余的弹出。可惜的是，这是不能自动发生的，我们必须注意正在设置的是什么标志，这样就不会偶然调用错误的 setf（）函数。例如，每一个数字基数有一个标志：十六进制，十进制和八进制。这些标志一起被指定为 ios::basefield。如果ios：：dec标志被设置而调用setf（ios::hex），将设置ios::hex标志，但不会清除ios::dec位，结果出现未被定义的方式。适当的方法是像这样调用 setf()的第二种形式：setf(ios::hex,ios::basefield)。这个函数首先清除ios::basefield里的所有位，然后设置ios::hex。这样，setf()的这个形式保证无论什么时候设置一个标志，这个组里的其他标志都会“弹出”。当然，所有这些操作由hex()操纵算子自动完成。所以不必了解这个类实现的内部细节，甚至不必关心它是一个二进制标志的设置。

以后将会看到有一个与set()有提供同样的功能操纵算子。

下面是标志组和它们的作用：

iso::basefield	作　　用
ios::dec	十进制格式整数值（十进制）（缺省基数）
ios::hex	十六进制格式整数值（十六进制）
ios::oct	八进制格式整数值（八进制）

ios::floatfield	作　　用
ios::scientific	科学记数法表示浮点数值，精度域指小数点后面的数字数目
ios::fixed	定点格式表示浮点数值，精度域指小数点后面的数字数目
“automatic”(Neither bit is set)	精度域指整个有效数字的数目

ios::adjustfield	作　　用
ios::left	左对齐，向右边填充字符
ios::right	右对齐，向左边填充字符
ios::internal	在任何引导符或基数指示符之后但在值之前添加填充字符

2. 域宽、填充字符和精度

一些内部变量，用于控制输出域的宽度，或当数据没有填入时，用作填充的字符，或控制打印浮点数的精度。它被与变量同名字的成员函数读和写。

function	作　　用
int ios::width()	读当前宽度（缺省值为0），用于插入和提取
int ios::width(int n)	设置宽度，返回以前的宽度
int ios::fill()	读当前填充字符（缺省值为空格）
int ios::fill(int n)	设置填充字符，返回以前的填充字符
int ios::precision()	读当前浮点数精度（缺省值为6）
int ios::precision(int n)	设置浮点精度，返回以前的精度；“精度”含义见 ios::floatfield表

填充和精度值是相当直观的，但宽度值需要一些解释。当宽度为0时，插入一个值将产生代表这个值所需字符的最小数。一个正值的宽度意味着插入一个值将产生至少与宽度一样多的字符。假如值小于字符宽度，填充字符用来填这个域。然而，这个值决不被截断。所以，如果打印123而宽度为2，我们仍将得到123。域宽标识了字符的最小数目。没有标识字符最大数目的办法。

宽度也是明显不同的，因为每个插入符或提取符可能受到它的值的影响，它被每个插入符或提取符重新设置为0。它不是一个真正的静态变量，而是插入符和提取符的一个隐含参数。如我们想有一个恒定的宽度，得在每一个插入或提取它之后调用width()。

6.7.2 例子

为了确信懂得怎样调用以前讨论过的所有函数，下面举一个全部调用这些函数的例子：

```
//: FORMAT.CPP -- Formatting functions
#include <fstream.h>
#define D(a) T << #a << endl; a
ofstream T("format.out");

main() {
```

```
D(int i = 47;)
D(float f = 2300114.414159;)
char* s = "Is there any more?";

D(T.setf(ios::unitbuf);)
D(T.setf(ios::stdio);)

D(T.setf(ios::showbase);)
D(T.setf(ios::uppercase);)
D(T.setf(ios::showpos);)
D(T << i << endl;) // Default to dec
D(T.setf(ios::hex, ios::basefield);)
D(T << i << endl;)
D(T.unsetf(ios::uppercase);)
D(T.setf(ios::oct, ios::basefield);)
D(T << i << endl;)
D(T.unsetf(ios::showbase);)
D(T.setf(ios::dec, ios::basefield);)
D(T.setf(ios::left, ios::adjustfield);)
D(T.fill('0');)
D(T << "fill char: " << T.fill() << endl;)
D(T.width(10);)
T << i << endl;
D(T.setf(ios::right, ios::adjustfield);)
D(T.width(10);)
T << i << endl;
D(T.setf(ios::internal, ios::adjustfield);)
D(T.width(10);)
T << i << endl;
D(T << i << endl;) // Without width(10)

D(T.unsetf(ios::showpos);)
D(T.setf(ios::showpoint);)
D(T << "prec = " << T.precision() << endl;)
D(T.setf(ios::scientific, ios::floatfield);)
D(T << endl << f << endl;)
D(T.setf(ios::fixed, ios::floatfield);)
D(T << f << endl;)
D(T.setf(0, ios::floatfield);) // Automatic
D(T << f << endl;)
D(T.precision(20);)
D(T << "prec = " << T.precision() << endl;)
D(T << endl << f << endl;)
D(T.setf(ios::scientific, ios::floatfield);)
D(T << endl << f << endl;)
D(T.setf(ios::fixed, ios::floatfield);)
```

```
  D(T << f << endl;)
  D(T.setf(0, ios::floatfield);) // Automatic
  D(T << f << endl;)

  D(T.width(10);)
  T << s << endl;
  D(T.width(40);)
  T << s << endl;
  D(T.setf(ios::left, ios::adjustfield);)
  D(T.width(40);)
  T << s << endl;

  D(T.unsetf(ios::showpoint);)
  D(T.unsetf(ios::unitbuf);)
  D(T.unsetf(ios::stdio);)
}
```

这个例子使用技巧建立一个跟踪文件，这样我们就能监控所发生的事情。宏D（a）使用预处理器“ stringizing ”把a转变为一个串打印出来。然后，宏D（a）反复处理a，使a产生作用。宏发送所有的信息到一个叫作T的文件中，这就是那个跟踪文件。输出是：

```
int i = 47;
float f = 2300114.414159;
T.setf(ios::unitbuf);
T.setf(ios::stdio);
T.setf(ios::showbase);
T.setf(ios::uppercase);
T.setf(ios::showpos);
T << i << endl;
+47
T.setf(ios::hex, ios::basefield);
T << i << endl;
+0X2F
T.unsetf(ios::uppercase);
T.setf(ios::oct, ios::basefield);
T << i << endl;
+057
T.unsetf(ios::showbase);
T.setf(ios::dec, ios::basefield);
T.setf(ios::left, ios::adjustfield);
T.fill('0');
T << "fill char: " << T.fill() << endl;
fill char: 0
T.width(10);
+470000000
T.setf(ios::right, ios::adjustfield);
T.width(10);
```

```
0000000+47
T.setf(ios::internal, ios::adjustfield);
T.width(10);
+000000047
T << i << endl;
+47
T.unsetf(ios::showpos);
T.setf(ios::showpoint);
T << "prec = " << T.precision() << endl;
prec = 6
T.setf(ios::scientific, ios::floatfield);
T << endl << f << endl;

2.300115e+06
T.setf(ios::fixed, ios::floatfield);
T << f << endl;
2300114.500000
T.setf(0, ios::floatfield);
T << f << endl;
2.300115e+06
T.precision(20);
T << "prec = " << T.precision() << endl;
prec = 20
T << endl << f << endl;

2300114.50000000020000000000
T.setf(ios::scientific, ios::floatfield);
T << endl << f << endl;

2.30011450000000020000e+06
T.setf(ios::fixed, ios::floatfield);
T << f << endl;
2300114.50000000020000000000
T.setf(0, ios::floatfield);
T << f << endl;
2300114.50000000020000000000
T.width(10);
Is there any more?
T.width(40);
0000000000000000000000Is there any more?
T.setf(ios::left, ios::adjustfield);
T.width(40);
Is there any more?0000000000000000000000
T.unsetf(ios::showpoint);
T.unsetf(ios::unitbuf);
T.unsetf(ios::stdio);
```

研究这个输出会使我们清楚地理解输入输出流格式化成员函数。

6.8 格式化操纵算子

就像我们在前面的例子中看到的一样，调用成员函数有点乏味。为使读和写更容易， C++提供了一套操纵算子以起到与成员函数同样的作用。

提供在IOSTREAM.H里的是不带参数的操纵算子。这些操纵算子包括 dec、oct和hex。它们各自更简明扼要地完成与 setf(ios::dec,ios::basefield)、setf(ios::oct,ios::basefield)和setf(ios::hex,ios::basefield)同样的任务。IOSTREAM.H [1] 还包括ws、endl、ends和flush以及如下所示的其他操纵算子：

操纵算子	作　　用
showbase	在打印一整数值时，标明数字基数（十进制，八进制和十六进制）；所用的格式能被C++编译器读出
noshowbase	
showpos	显示正值符号加（+）
noshowpos	
uppercase	显示代表十六进制值的大写字母A-F以及科学记数法中的E
nouppercase	
showpoint	表明浮点数值的小数点和后面的零
noshowpoint	
skipws	跳过输入中的空白字符
noshipws	
left	左对齐，右填充
right	右对齐，左填充
internal	在引导符或基数指示符和值之间填充
scientific	使用科学记数法
fixed	setprecision()或ios::precision()设置小数点后面的位数

带参数的操纵算子

如果正在使用带参数的操纵算子，必须也包含头文件IOMANIP.H。这包含了解决建立带参数操纵算子所遇到的一般问题的代码。另外，它有六个预定义的操纵算子：

操纵算子	作　　用
setiosflags(fmtflags n)	设置由n指定的格式标志；设置一直起作用直到下一个变化为止，像ios::setf()一样
resetiosflags(fmtflags n)	清除由n指定的格式标志。设置一直起作用直到下一个变化为止，像ios::unsetf()一样
setbase(base n)	把基数改成n，这里n取10、8或16（任何别的值结果为0）。如果n是0，输出基数为10，但输入使用C约定：10是10，010是8而0xf是15。我们还是使用dec、oct和hex输出为好
setfill(char n)	把填充字符改成n，像ios∷fill（）一样
setprecision(int n)	把精度改成n，像ios∷precision（）一样
setw(int n)	把域宽改成n，像ios∷width（）一样

如果正在使用很多的插入符，我们会看到这是怎样清理的。作为一个例子，下面是用操纵

[1] 这些仅在修改库中出现，老的输入输出流实现中没包括它们。

算子对前面程序的重写（宏已被去掉以使其更容易阅读）：

```
//: MANIPS.CPP -- FORMAT.CPP using manipulators
#include <fstream.h>
#include <iomanip.h>

main() {
  ofstream T("trace.out");
  int i = 47;
  float f = 2300114.414159;
  char* s = "Is there any more?";

  T << setiosflags(
         ios::unitbuf | ios::stdio
         | ios::showbase | ios::uppercase
         | ios::showpos
       );
  T << i << endl; // Default to dec
  T << hex << i << endl;
  T << resetiosflags(ios::uppercase)
    << oct << i << endl;
  T.setf(ios::left, ios::adjustfield);
  T << resetiosflags(ios::showbase)
    << dec << setfill('0');
  T << "fill char: " << T.fill() << endl;
  T << setw(10) << i << endl;
  T.setf(ios::right, ios::adjustfield);
  T << setw(10) << i << endl;
  T.setf(ios::internal, ios::adjustfield);
  T << setw(10) << i << endl;
  T << i << endl; // Without setw(10)

  T << resetiosflags(ios::showpos)
    << setiosflags(ios::showpoint)
    << "prec = " << T.precision() << endl;
  T.setf(ios::scientific, ios::floatfield);
  T << f << endl;
  T.setf(ios::fixed, ios::floatfield);
  T << f << endl;
  T.setf(0, ios::floatfield); // Automatic
  T << f << endl;
  T << setprecision(20);
  T << "prec = " << T.precision() << endl;
  T << f << endl;
  T.setf(ios::scientific, ios::floatfield);
  T << f << endl;
  T.setf(ios::fixed, ios::floatfield);
```

```
  T << f << endl;
  T.setf(0, ios::floatfield); // Automatic
  T << f << endl;

  T << setw(10) << s << endl;
  T << setw(40) << s << endl;
  T.setf(ios::left, ios::adjustfield);
  T << setw(40) << s << endl;

  T << resetiosflags(
         ios::showpoint | ios::unitbuf
         | ios::stdio
       );
}
```

许多的多重语句已被精简成单个的链插入。注意调用 setiosflags()和resetiosflags()，在这两个函数里，标志被按“位OR”运算成一个。在前面的例子里，这个工作是由 setf()和unsetf()完成的。

6.9 建立操纵算子

（注意：这部分包含一些以后章节中才介绍的内容）有时，我们可能想建立自己的操纵算子，这是相当简单的。一个像endl这样的不带参数的操纵算子只是一个函数，这个函数把一个ostream引用作为它的参数。（引用是一种不同的参数传送方式，在第 10章中讨论）对endl的声明是：

```
ostream& endl(ostream&)；
```

现在，当我们写：

```
cout<<"howdy" <<endl；
```

endl产生函数的地址。这样，编译器问：“有能被我调用的把函数的地址作为它的参数的函数吗？”确实有一个这样的函数，是在IOSTREAM.H里预先定义的函数；它被称作“应用算子”。这个应用算子调用这个函数，把ostream对象作为一个参数传送给这个函数。

不必知道应用算子怎样建立我们自己的操纵算子；我们只要知道应用算子存在就行了。下面是建立一个操纵算子的例子，这个操纵算子叫nl，它产生一个换行而不刷新这个流：

```
//: NL.CPP -- Creating a manipulator
#include <iostream.h>

ostream& nl(ostream& os) {
  return os << '\n';
}

main() {
  cout << "newlines" << nl << "between" << nl
       << "each" << nl << "word" << nl;
}
```

表达式

```
os<<'\n';
```

调用一个返回os的函数，它是从nl中返回的[1]。

人们经常认为，如上所示的nl比使用endl要好，因为后者总是清空输出流，这可能引起执行故障。

效用算子

正如我们已看到的，零参数操纵算子相当容易建立。但是如果建立带参数的操纵算子又怎样呢？要这样做，输入输出流有一个相当麻烦且易混淆的方法。但是 Jerry Schwarz，输入输出流库的建立者，提出一个方案[2]，他称之为效用算子。一个效用算子是一个简单的类，这个类的构造函数与工作在这个类里的一个重载操作符“ << ”一起执行想要的操作。下面是一个有两个效用算子的例子。第一个输出是一个被截断的字符串，第二个打印出一个二进制数（定义一个重载操作符“ << ”的过程将在第11章讨论）：

```
//: EFFECTOR.CPP -- Jerry Schwarz's "effectors"
#include<iostream.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <limits.h> // ULONG_MAX

// Put out a portion of a string:
class fixw {
  char* s;
public:
  fixw(const char* S, int width);
  ~fixw();
  friend ostream& operator<<(ostream&, fixw&);
};

fixw::fixw(const char* S, int width) {
  s = (char*)malloc(width + 1);
  assert(s);
  strncpy(s, S, width);
  s[width] = 0; // Null-terminate
}

fixw::~fixw() { free(s); }

ostream& operator<<(ostream& os, fixw& fw) {
  return os << fw.s;
}
```

[1] 把n1放入头文件之前，应该把它变成内联函数（见第8章）。

[2] 在私人谈话中。

```
// Print a number in binary:
typedef unsigned long ulong;

class bin {
  ulong n;
public:
  bin(ulong N);
  friend ostream& operator<<(ostream&, bin&);
};

bin::bin(ulong N) { n = N; }

ostream& operator<<(ostream& os, bin& b) {
  ulong bit = ~(ULONG_MAX >> 1); // Top bit set
  while(bit) {
    os << (b.n & bit ? '1' : '0');
    bit >>= 1;
  }
  return os;
}
main() {
  char* string =
    "Things that make us happy, make us wise";
  for(int i = 1; i <= strlen(string); i++)
    cout << fixw(string, i) << endl;
  ulong x = 0xFEDCBA98UL;
  ulong y = 0x76543210UL;
  cout << "x in binary: " << bin(x) << endl;
  cout << "y in binary: " << bin(y) << endl;
}
```

fixw的构造函数产生char*参数的一个缩短的副本，析构函数释放产生这个副本的内存。重载操作符“ << ”取第二个参数的内容，即fixw对象，并把它插入第一个参数ostream，然后返回ostream，所以它可以用在一个链接表达式里。下面的表达式里用fixw时：

```
cout <<fixw(string,i)<<endl；
```

一个临时对象通过调用fixw构造函数被建立了，那个临时对象被传送给操作符“ << ”。带参数的操纵算子产生作用了。

bin 效用算子依赖于这样的事实：对一个无符号数向右移位，把零移成高位。 ULONG_MAX（最大的长型值unsigned long，来自标准包含文件LIMITS.H）用来产生一个带高位设置的值，这个值移过正被讨论的数（通过移位），屏蔽每一位。

起初，这个技术上的问题是：一旦为char*建立一个叫fixw的类，或为unsign long建立一个叫bin的类，没有别的人能为他们的类型建立一个不同的 fixw类或bin类。然而，有了名字空间（在第9章），这个问题被解决了。

6.10 输入输出流实例

本节，将看到怎样处理本章学到的所有知识的一些例子。虽然存在能操纵字节的很多工具（像unix中的sed和awk这样的流编辑器可能是广为人知的，而文本编辑器也属于此类），但它们一般都有一些限制。sed和awk可能比较慢，而且仅能处理向前序列里的行。文本编辑器通常需要人的交互作用或至少要学一种专有的宏语言。用输入输出流写的程序就没有任何此类限制：这些程序运行快，可移植而且很灵活。输入输出流是工具箱里的非常有用的一个工具。

6.10.1 代码生成

第一个例子涉及程序的产生，比较巧的是，这个程序也符合本书的格式。在开发代码时保证供快速和连贯性。第一个程序建立一个文件保存 main（）（假定它带有非命令行参数并且使用输入输出流库）：

```
//: MAKEMAIN.CPP -- Create a shell main() File
#include <fstream.h>
#include <strstrea.h>
#include <assert.h>
#include <string.h>
#include <ctype.h>

main(int argc, char* argv[]) {
  assert(argc == 2);
  // Don't replace it if it exists:
  ofstream mainfile(argv[1], ios::noreplace);
  assert(mainfile);
  istrstream name(argv[1]);
  ostrstream CAPname;
  char c;
  while(name.get(c))
    CAPname << char(toupper(c));
  CAPname << ends;
  mainfile << "//:"  << ' ' << CAPname.rdbuf()
    << " -- " << endl
    << "#include <iostream.h>" << endl
    << endl
    << "main() {" << endl << endl
    << "}" << endl;
}
```

这个文件被打开，使用ios::noreplace，以保证不会偶然地覆盖一个现成文件。然后用命令行中的那个参数来建立一个 istrstream。这样，字符每次一个地被提取。有了标准 C库宏toupper()，这些字符可被转变成大写字母。这个变量返回一个 int，这样，它必须很明确地转换给char。此名字用在头一行里，后跟产生文件的余项。

1. 维护类库资源

第二个例子执行一个更复杂和更有用的任务。总之，建立一个类时，要想想库术语，而且

要在类声明中创建一个头文件NAME.H和一个名叫NAME.CPP的文件——成员函数在这个文件里实现。这些文件有一定的要求：一个特别的代码标准（这儿显示的程序是用这本书里的代码格式），而且这个头文件的声明一般被一些预处理器说明所包围，目的是避免类的多重说明。（多重说明使编译器混淆——编译器不知道我们要使用哪个类。这些类可能是不同的，所以编译器发出一个出错信息）。

这个例子建立一对新的头——实现文件，不然就修改现存的文件。如果这对文件已存在，它检查这对文件并潜在地修改这对文件，但是如果这对文件不存在，它用合适的格式建立这对文件：

```
//: CPPCHECK.CPP -- Configures .H & .CPP files
// To conform to style standard.
// Tests existing files for conformance
#include <fstream.h>
#include <strstrea.h>
#include <string.h>
#include <ctype.h>
#include <assert.h>
#define SZ 40  // Buffer sizes
#define BSZ 100

main(int argc, char* argv[]) {
  assert(argc == 2); // File set name
  enum bufs { base, header, implement,
    Hline1, guard1, guard2, guard3,
    CPPline1, include, bufnum };
  char b[bufnum][SZ];
  ostrstream osarray[] = {
    ostrstream(b[base], SZ),
    ostrstream(b[header], SZ),
    ostrstream(b[implement], SZ),
    ostrstream(b[Hline1], SZ),
    ostrstream(b[guard1], SZ),
    ostrstream(b[guard2], SZ),
    ostrstream(b[guard3], SZ),
    ostrstream(b[CPPline1], SZ),
    ostrstream(b[include], SZ),
  };
  osarray[base] << argv[1] << ends;
  // Find any '.' in the string using the
  // Standard C library function strchr():
  char* period = strchr(b[base], '.');
  if(period) *period = 0; // Strip extension
  // Force to upper case:
  for(int i = 0; b[base][i]; i++)
    b[base][i] = toupper(b[base][i]);
  // Create file names and internal lines:
```

```
osarray[header] << b[base] << ".H" << ends;
osarray[implement] << b[base] << ".CPP" << ends;
osarray[Hline1] << "//:" << ' ' << b[header]
  << " -- " << ends;
osarray[guard1] << "#ifndef " << b[base]
                << "_H_" << ends;
osarray[guard2] << "#define " << b[base]
                << "_H_" << ends;
osarray[guard3] << "#endif // " << b[base]
                << "_H_" << ends;
osarray[CPPline1] << "//:" << ' '
                  << b[implement]
                  << " -- " << ends;
osarray[include] << "#include \""
                 << b[header] << "\"" <<ends;
// First, try to open existing files:
ifstream existh(b[header]),
         existcpp(b[implement]);
if(!existh) { // Doesn't exist; create it
  ofstream newheader(b[header]);
  newheader << b[Hline1] << endl
    << b[guard1] << endl
    << b[guard2] << endl << endl
    << b[guard3] << endl;
}
if(!existcpp) { // Create cpp file
  ofstream newcpp(implement);
  newcpp << b[CPPline1] << endl
    << b[include] << endl;
}
if(existh) { // Already exists; verify it
  strstream hfile; // Write & read
  ostrstream newheader; // Write
  hfile << existh.rdbuf() << ends;
  // Check that first line conforms:
  char buf[BSZ];
  if(hfile.getline(buf, BSZ)) {
    if(!strstr(buf, "//:") ||
      !strstr(buf, b[header]))
     newheader << b[Hline1] << endl;
  }
  // Ensure guard lines are in header:
  if(!strstr(hfile.str(), b[guard1]) ||
     !strstr(hfile.str(), b[guard2]) ||
     !strstr(hfile.str(), b[guard3])) {
```

```
      newheader << b[guard1] << endl
        << b[guard2] << endl
        << buf
        << hfile.rdbuf() << endl
        << b[guard3] << endl << ends;
    } else
      newheader << buf
        << hfile.rdbuf() << ends;
    // If there were changes, overwrite file:
    if(strcmp(hfile.str(),newheader.str())!=0){
      existh.close();
      ofstream newH(b[header]);
      newH << "//@//" << endl // change marker
        << newheader.rdbuf();
    }
    delete hfile.str();
    delete newheader.str();
  }
  if(existcpp) { // Already exists; verify it
    strstream cppfile;
    ostrstream newcpp;
    cppfile << existcpp.rdbuf() << ends;
    char buf[BSZ];
    // Check that first line conforms:
    if(cppfile.getline(buf, BSZ))
      if(!strstr(buf, "//:") ||
         !strstr(buf, b[implement]))
        newcpp << b[CPPline1] << endl;
    // Ensure header is included:
    if(!strstr(cppfile.str(), b[include]))
        newcpp << b[include] << endl;
      // Put in the rest of the file:
      newcpp << buf << endl; // First line read
      newcpp << cppfile.rdbuf() << ends;
      // If there were changes, overwrite file:
      if(strcmp(cppfile.str(),newcpp.str())!=0){
        existcpp.close();
        ofstream newCPP(b[implement]);
        newCPP << "//@//" << endl // change marker
          << newcpp.rdbuf();
      }
      delete cppfile.str();
      delete newcpp.str();
    }
  }
```

这个例子需要不同缓冲区中的串格式化。不是建立单个命名的缓冲区和 ostrstream对象，而是在enum bufs缓冲区中建立 一组名字。因而要建立两个数组：一个字符缓冲区数组和一个从字符缓冲区里建立的ostrstream对象数组。注意在char缓冲区b的二维数组定义里， char数组的数目是由bufnum决定的， bufnum是bufs中的最后一个枚举常量。当建立一个枚举变量时，编译器赋一个整数值给所有那些标明从零开始的 enum，所以bufnum的唯一目的是成为统计buf中枚举常量数目的计数器。b中每一个串的长度是SZ。

枚举变量里的名字是： base，即大写字母的不带扩展名的基文件名； header，即头文件名；implement，即实现文件名（CPP）；Hline1，即头文件第一行框架； guard1 、guard2和guard3，即头文件里的“保护”行（阻止多重包含）； CPPlinel，即CPP文件的第一行框架；include，即包含头文件的CPP文件的行。

osarray是一个通过集合初始化和自动计数建立起来的 ostrstream对象数组。当然，这是带两个参数（缓冲区地址和大小）形式的 ostrstream构造函数，所以构造函数调用必须相应地建立在集合初始化表里。利用 bufs枚举常量， b的适当数组元素结合到相应的 osarray对象。一旦数组被建立，利用枚举常量就可选择数组里的对象，作用是填充相应的 b元素。我们可看到，每个串是怎样建立在ostrstream数组定义后面的行里的。

一旦串被建立，程序试图打开头文件和 CPP文件的现行版本作为ifstreams。如果用操作符“！”测试对象而这个文件不存在，测试将失败。如果头文件或实现文件不存在，利用以前建立的文本的适当的行来建立它。

如果文件确实存在，那么这些文件后面跟着适当的格式，这是有保证的。在这两种情况下，一个strstream被建立而且整个文件被读进；然后第一行被读出并被检查，看看这一行是否包含一个“//:”和文件名以确信它跟在格式的后面。这是由标准 C库函数strstr()完成的。如果第一行不符合，较早时建立的内容被插进一个已建好的 ostrstream中，目的是保存被编辑的那个文件。

在头文件里，整个文件被搜索（再次使用 strstr()函数）以确保它包含三个“保护”行；如果没有包含这三行，要插入它们。检查实现文件，看看包含头文件那一行是否存在（虽然编译器有效地保证它的存在）。

在两种情况下，比较原始文件（在它的 strstream里）和被编辑的文件（在 ostrstream里），看看它们是否有变化。如果有变化，现存文件被关闭， 一个新的 ofstream对象被建立，目的是覆盖这个现存文件。一个特别的变化标志被添加到开始处之后， ostrstream被输出到这个文件，所以可使用一文本搜索程序，通过检查快速发现产生另外变化的文件。

2. 检测编译器错误

这本书里的所有代码都已经设计好了，没有编译错误。任何产生编译错误的代码行都由特别注释顺序符“//！”注解出来。下面的程序将移去这些特别注解并在原处添加一个已编号的注解。这样，运行编译器时，它应该产生出错信息。当编译所有文件时，应该看到所有的号码。它也可以添加修改过的行到一个特别文件里，这样可容易定位任何不产生错误的行：

```
//: SHOWERR.CPP -- Un-comment error generators
#include <fstream.h>
#include <strstrea.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <assert.h>
```

```
char* marker = "//!";

char* usage =
"usage: showerr filename chapnum\n"
"where filename is a C++ source file\n"
"and chapnum is the chapter name it's in.\n"
"Finds lines commented with //! and removes\n"
"comment, appending //(#) where # is unique\n"
"across all files, so you can determine\n"
"if your compiler finds the error.\n"
"showerr /r\n"
"resets the unique counter.";

// File containing error number counter:
char* errnum = "..\\errnum.txt";
// File containing error lines:
char* errfile = "..\\errlines.txt";
ofstream errlines(errfile,ios::app);

main(int argc, char* argv[]) {
  if(argc < 2) {
    cerr << usage << endl;
    return 1;
  }
  if(argv[1][0] == '/' || argv[1][0] == '-') {
    // Allow for other switches:
    switch(argv[1][1]) {
      case 'r': case 'R':
        cout << "reset counter" << endl;
        remove(errnum); // Delete files
        remove(errfile);
        return 0;
      default:
        cerr << usage << endl;
        return 1;
    }
  }
  char* chapter = argv[2];
  strstream edited; // Edited file
  int counter = 0;
  {
    ifstream infile(argv[1]);
    assert(infile);
    ifstream count(errnum);
    if(count) count >> counter;
    int linecount = 0;
```

```
    #define sz 255
    char buf[sz];
    while(infile.getline(buf, sz)) {
      linecount++;
      // Eat white space:
      int i = 0;
      while(isspace(buf[i]))
      i++;
    // Find marker at start of line:
    if(strstr(&buf[i], marker) == &buf[i]) {
      // Erase marker:
      memset(&buf[i], ' ', strlen(marker));
      // Append counter & error info:
      ostrstream out(buf, sz, ios::ate);
      out << "//(" << ++counter << ") "
          << "Chapter " << chapter
          << " File: " << argv[1]
          << " Line " << linecount << endl
          << ends;
        edited << buf;
      errlines << buf; // Append error file
    } else
      edited << buf << "\n"; // Just copy
  }
 } // Closes files
 ofstream outfile(argv[1]); // Overwrites
 outfile << edited.rdbuf();
 ofstream count(errnum); // Overwrites
 count << counter; // Save new counter
}
```

这个maker指针可被我们的一种选择所代替。

每个文件每次读出一行，在每一行中搜索出现在行开头的 maker。这个行被修改并放进错误行表、放进strstream edited里。当整个文件被处理完时，它被关闭（通过到达范围的末端），重新作为一个输出文件打开， edited被灌进这个文件。注意计数器被保存在内部文件里。所以下一次调用这个程序时，继续由计数器按顺序计数。

6.10.2 一个简单的数据记录

这个例子显示了记录数据到磁盘上而后检索数据并作处理的一种途径。这个例子的意思是产生一个海洋各处温度——深度的曲线图。为保存数据，要用到一个类：

```
//: DATALOG.H -- Datalogger record layout
#ifndef DATALOG_H_
#define DATALOG_H_
#include <time.h>
#include <iostream.h>
```

```
#define BSZ 10

class datapoint {
  tm Tm; // Time & day
  // Ascii degrees (*) minutes (') seconds ("):
  char Latitude[BSZ], Longitude[BSZ];
  double Depth, Temperature;
public:
  tm Time(); // read the time
  void Time(tm T); // set the time
  const char* latitude(); // read
  void latitude(const char* l); // set
  const char* longitude(); // read
  void longitude(const char* l); // set
  double depth(); // read
  void depth(double d); // set
  double temperature(); // read
  void temperature(double t); // set
  void print(ostream& os);
};
#endif // DATALOG_H_
```

这个存取函数为每个数据成员提供受控制的读和写。 printf()函数用一个可读的形式格式化datapoint到一个ostream对象中（printf()的参数），下面是一个定义文件：

```
//: DATALOG.CPP -- Datapoint member functions
#include "..\5\datalog.h"
#include <iomanip.h>
#include <string.h>

tm datapoint::Time() { return Tm; }

void datapoint::Time(tm T) { Tm = T; }

const char* datapoint::latitude() {
  return Latitude;
}

void datapoint::latitude(const char* l) {
  Latitude[BSZ - 1] = 0;
  strncpy(Latitude, l, BSZ - 1);
}

const char* datapoint::longitude() {
  return Longitude;
}
```

```
void datapoint::longitude(const char* l) {
  Longitude[BSZ - 1] = 0;
  strncpy(Longitude, l, BSZ - 1);
}

double datapoint::depth() { return Depth; }

void datapoint::depth(double d) { Depth = d; }

double datapoint::temperature() {
  return Temperature;
}

void datapoint::temperature(double t) {
  Temperature = t;
}

void datapoint::print(ostream& os) {
  os.setf(ios::fixed, ios::floatfield);
  os.precision(4);
  os.fill('0'); // Pad on left with '0'
  os << setw(2) << Time().tm_mon << '\\'
     << setw(2) << Time().tm_mday << '\\'
     << setw(2) << Time().tm_year << ' '
     << setw(2) << Time().tm_hour << ':'
     << setw(2) << Time().tm_min << ':'
     << setw(2) << Time().tm_sec;
  os.fill(' '); // Pad on left with ' '
  os << " Lat:" << setw(9) << latitude()
     << ", Long:" << setw(9) << longitude()
     << ", depth:" << setw(9) << depth()
     << ", temp:" << setw(9) << temperature()
     << endl;
}
```

在printf()函数里，调用setf()引起浮点数以定点精度的形式输出，而精度函数 precision()设置4位小数。

域里的数据缺省向右对齐，时间信息由时、分和秒各两位数字组成，这样。在每种情况下，宽度由setw()函数设置为2。（记住域宽的任何变化影响下次输出操作，所以setw()函数必须配给每个输出）。但是首先如果值小于10，填充字符被设置为“ 0 ”，在值的左边放一个零。以后它又被设置为空格。

纬度和经度是零终止符域，它保存度（这里‘ * ’表示度）、分（`）和秒（``）的信息。如我们愿意的话，当然能设计出一个更有效的存储方案。

1. 生成测试数据

下面是一个程序，这个程序（用 write()）建立一个二进制形式的测试数据文件 ,并且用

datapoint::print()建立另一个ASCⅡ形式的文件。我们也可以把它打印到屏幕上，但是在文件形式里更容易检测：

```
//: DATAGEN.CPP -- Test data generator
#include <fstream.h>
#include <stdlib.h>
#include <string.h>
#include "..\5\datalog.h"

main() {
  ofstream data("data.txt");
  ofstream bindata("data.bin", ios::binary);
  time_t timer = time(NULL);  // Get time
  // Seed random generator:
  srand((unsigned)timer);
  for(int i = 0; i < 100; i++) {
    datapoint d;
    // Convert date/time to a structure:
    d.Time(*localtime(&timer));
    timer += 55; // Reading each 55 seconds
    d.latitude("45*20'31\"");
    d.longitude("22*34'18\"");
    // Zero to 199 meters:
    double newdepth  = rand() % 200;
    double fraction = rand() % 100 + 1;
    newdepth += double(1) / fraction;
    d.depth(newdepth);
    double newtemp = 150 + rand()%200; //Kelvin
    fraction = rand() % 100 + 1;
    newtemp += (double)1 / fraction;
    d.temperature(newtemp);
    d.print(data);
    bindata.write((unsigned char*)&d,
                  sizeof(d));
  }
}
```

文件DATA.TXT作为一个ASCⅡ文件用通常的方法建立了。而DATA.BIN有标志ios::binary，用以告诉构造函数把它设置成二进制文件。

标准C库函数time(),当带一个零参数调用它时，返回当前时间为一个time_t值，它是自1970年1月1日00：00：00 GMT后的秒数。当前时间是用标准C库函数scrand()为随机数产生器设置种子的最方便的方法，这里就是这样做的。

有时候，存储时间的更方便的方法是一个tm结构，它含有时间和日期的每个元素，而时间和日期被分成如下所示的构成成分：

```
struct tm {
   int tm_sec; // 0-59 seconds
```

```
    int tm_min; // 0-59 minutes
    int tm_hour; // 0-23 hours
    int tm_mday; // day of month
    int tm_mon; // 1-12 months
    int tm_year; // calendar year
    int tm_wday; // Sunday == 0, etc.
    int tm_yday; // 0-365 day of year
    int tm_isdst; // daylight savings?
};
```

为把以秒计数的时间转换成tm格式的本地时间，利用标准C库localtime()函数，它取时间秒数并返回一个指针指向tm中的结果。然而，tm是localtime()函数里面的一个静态结构，每当localtime()被调用时，localtime()被重写。为把内容拷进datapoint中的tm struct中，我们可能认为必须逐个拷贝每个元素。然而，我们所必须做的是一个结构分配，其余的由编译器来做，这意味着右边的一定是一个结构，不是一个指针，所以，localtime()的结果被间接引用。想要得到的结果是：

```
d.Time=*localtime(&timer)；
```

在这之后，timer增加了55秒，与读之间有一个有趣的间隔。

使用的纬度和经度是定点值，这个值表明在某一位置的一组读出值。深度和温度是由标准C库rand()函数产生的，这个函数返回零和常数RAND_MAX之间的一个伪随机数。为把这个数放进希望得到的范围里，使用模操作符%和范围的上界。这些数是整型的；为添加小数部分，对函数rand()做第二次调用，得到的值加一以后取倒数（加一是为了防止除数为零的错误）。

事实上，文件DATA.BIN作为数据容器在程序里使用，即使这个容器存在于磁盘上而不存在RAM中。为了以二进制形式发送这些数据到磁盘上，write()被使用。第一个参数是源块的开始地址——注意它必须指派为unsigned char*，因为这正是这个函数所期望的。第二参数代表要写的字节数，就是datapoint对象的大小。由于没有指针包含在datapoint里，因此向盘中写入对象时不会出现问题。如果对象更复杂，我们必须实现一个顺序化方案（大多数类库厂家已在里面建立了某种顺序化）。

2. 检验和观察数据

为检查以二进制格式存储的数据的有效性，数据从盘中读出并被放进文本文件DATA2.TXT中，所以这个文件可与DATA.TXT比较以供检验。在下面的程序里，我们会看到这个数据恢复是多么简单。在测试文件被建立后，记录在用户命令上被读出。

```
//: DATASCAN.CPP -- Verify and view logged data
#include <fstream.h>
#include <strstrea.h>
#include <iomanip.h>
#include <assert.h>
#include "..\5\datalog.h"

main() {
  ifstream bindata("data.bin", ios::binary);
  assert(bindata); // Make sure it exists
  // Create comparison file to verify data.txt:
```

```
  ofstream verify("data2.txt");
  datapoint d;
  while(bindata.read((unsigned char*)&d,
                     sizeof d))
    d.print(verify);
  bindata.clear(); // Reset state to "good"
  // Display user-selected records:
  int recnum = 0;
  // Left-align everything:
  cout.setf(ios::left, ios::adjustfield);
  // Fixed precision of 4 decimal places:
  cout.setf(ios::fixed, ios::floatfield);
  cout.precision(4);
  for(;;) {
    bindata.seekg(recnum* sizeof d, ios::beg);
    cout << "record " << recnum << endl;
    if(bindata.read((unsigned char*)&d,
                    sizeof d)) {
      cout << asctime(&(d.Time()));
      cout << setw(11) << "Latitude"
           << setw(11) << "Longitude"
           << setw(10) << "Depth"
           << setw(12) << "Temperature"
           << endl;
      // Put a line after the description:
      cout << setfill('-') << setw(43) << '-'
           << setfill(' ') << endl;
      cout << setw(11) << d.latitude()
           << setw(11) << d.longitude()
           << setw(10) << d.depth()
           << setw(12) << d.temperature()
           << endl;
    } else {
      cout << "invalid record number" << endl;
      bindata.clear(); // Reset state to "good"
    }
    cout << endl
         << "enter record number, x to quit:";
    char buf[10];
    cin.getline(buf, 10);
    if(buf[0] == 'x') break;
    istrstream input(buf, 10);
    input >> recnum;
  }
}
```

ifstream bindata是从文件DATA.BIN中产生并作为一个二进制文件建立起来的，带有

ios::nocreate标志。如果这个文件不存在，这个标志被设置导致函数 assert()失败。函数read()说明读出一个单个记录并把它直接放进datapoint d。（假如datapoint包含指针，这将产生无意义的指针值）。到文件尾时，函数read()的作用是设置bindata的failbit。它将导致while语句失败。然而，在这一点上，不能移回“取”指针并读更多的记录，因为流的状态不允许进一步读出。所以clear()函数被调用，重新设置failbit。

一旦记录从盘里读进，我们就可以做我们想做的任何事情，如执行计算或作图。这里，它被显示出来以进一步练习关于输入输出流格式方面的知识。

程序的其余部分显示用户选择的一个记录号（由 recnum代表）。像以前一样，精度是固定在4个小数的地方。但这时都是左对齐的。

这个输出的格式看来与以前不同：

```
record 0
Tue Nov 16 18:15:49 1993
Latitude    Longitude   Depth      Temperature
----------------------------------------
45*20'31"   22*34'18"   186.0172   269.0167
```

为确信标号和数据栏纵向对齐，利用 setw()，标号被放入与栏同样的宽度域内。通过设置填字符‘－’，设置希望得到的行宽而且输出单个的‘－’，这样，行间隔符产生了。

如果read()失败，我们将在else部分结束，这部分告诉用户记录数是无效的。然后，由于failbit被设置，必须调用clear()重新设置它。这样，下一个read()是成功的（假定它在正确的范围内）。

当然，也可以打开二进制数据文件来读和写。这样可以检索记录，修改它们并把它们写回到同样的位置，建立了flat-file数据库管理系统。就在我第一次做程序设计工作时，我也不得不建立乏味的文件数据库管理系统 DBMS——在Apple Ⅱ上用BASIC。它花费了几个月时间，然而，现在这样做只需几分钟。当然，现在使用一个封装的 DBMS会更有意义，但是用C++和输入输出流，仍可做在实验室里需要做的所有低级操作。

6.11 小结

这一章给出了关于输入输出流类库的一个相当完整的介绍。很可能这就是用输入输出流建立程序所必需的。（在以后的章节里，我们会看到向我们自己的类里添加输入输出流功能的简单例子。）然而，应该知道，输入输出流还有一些不常使用的其他性能，可通过查看输入输出流头文件和读我们自己编译器中的关于输入输出流的文档来发现这些性能。

6.12 练习

1) 通过创建一个叫in 的ifstream对象来打开一个文件。创建一个叫os的ostrstream对象，并通过rdbuf()成员函数把整个内容读进ostrstream。用str()函数取出os的char*地址，并利用标准C toupper() 宏使文件里每个字符大写。把结果写到一新的文件中，并删除由 os分配的内存。

2) 创建一个能打开文件（命令行中的第一个参数）的程序，并从中搜索一组字中的任何一个（命令行中其余的参数）。每次，读入一行输入并打印出与之匹配的行（带行数）。

3) 写一个在所有源代码文件的开始处添加版权注意事项的程序。只需对练习 1）稍作修改。

4) 用你最喜爱的文本搜索程序（如grep）输出包含一特殊模式的所有文件名字（仅是名字）。重定向输出到一个文件中。写一个用那个文件里的内容产生批处理文件的程序，这个批处理文件对每个由这个搜索程序找到的文件调用你的编辑器。

第7章 常 量

常量概念的建立（由关键字const表示）允许程序员在变化和不变化之间划一条界线。

在C++程序设计项目中提供了安全性和可控性。自从常量问世以来，它就有着很多不同的作用。与此同时，它在C语言中的意义又不一样。开始时，看起来容易混淆。在这一章里，我们将介绍什么时候、为什么和怎样使用关键字 const。最后，讨论volatile,它是const的“兄弟”（因为它们都关系到是否变化，而且语法也一样）。

const的最初动机是取代预处理器 #defines进行值替代。从此它曾被用于指针、函数变量、返回类型、类对象及其成员函数。所有这些用法都稍有区别，但它们在概念上是一致的，我们将在以下各节中说明这些用法。

7.1 值替代

用C语言进行程序设计时，预处理器可以不受限制地建立宏并用它来替代值。因为预处理器只做文本替代，它既没有类型检查思想，也没有类型检查工具，所以预处理器的值替代会产生一些微小的问题，这些问题在C++中可通过使用const而避免。

C语言中预处理器用值替代名字的典型用法是这样的：

```
#define BUFSIZE 100
```

BUFSIZE是一个名字，它不占用存储空间且能放在一个头文件里，目的是为使用它的所有编译单元提供一个值。用值替代而不是用所谓的“不可思议的数”，这对于支持代码维护是非常重要的。如果代码中用到不可思议的数，读者不仅不清楚这个数字来自哪里，而且也不知道它代表什么，进而，当决定改变一个值时，程序员必须执行手动编辑，而且还不能跟踪以保证没有漏掉其中的一个。

多数情况，BUFSIZE的工作方式与普通变量一样但也不都如此；而且这种方法还存在一个类型问题。这就会隐藏一些很难发现的错误。 C++用const把值替代带进编译器领域来解决这些问题。可以这样写：

```
const bufsize=100；
```

或用更清楚的形式：

```
const int bufsize=100；
```

这样就可以在任何编译器需要知道这个值的地方使用 bufsize，同时它还可以执行常量折叠，也就是说，编译器在编译时可以通过必要的计算把一个复杂的常量表达式缩减成简单的。这一点在数组定义里显得尤其重要：

```
char buf[bufsize]；
```

我们可以为所有的内部数据类型（ char、int、float和double型）以及由它们所定义的变量（也可以是类的对象，这将在以后章节里讲到）使用限定符 const。我们应该完全用 const取代#define的值替代。

7.1.1 头文件里的const

与使用#define一样，使用const必须把const定义放进头文件里。这样，通过包含头文件，

可把const定义单独放在一个地方并把它分配给一个编译单元。C++中的const默认为内部连接，也就是说，const仅在const被定义过的文件里才是可见的，而在连接时不能被其他编译单元看到。当定义一个常量（const）时，必须赋一个值给它，除非用extern作了清楚的说明：

```
extern const bufsize；
```

虽然上面的extern强制进行了存储空间分配（另外还有一些情况，如取一个 const的地址，也要进行存储空间分配），但是C++编译器通常并不为const分配存储空间，相反它把这个定义保存在它的符号表里。当const被使用时，它在编译时会进行常量折叠。

当然，绝对不为任何const分配存储是不可能的，尤其对于复杂的结构。这种情况下，编译器建立存储，这会阻止常量折叠。这就是 const为什么必须默认内部连接，即连接仅在特别编译单元内的原因；否则，由于众多的 const在多个cpp文件内分配存储，容易引起连接错误，连接程序在多个对象文件里看到同样的定义就会“抱怨”了。然而，因为 const默认内部连接，所以连接程序不会跨过编译单元连接那些定义，因此不会有冲突。对于在大量场合使用的内部数据类型，包括常量表达式，编译器都能执行常量折叠。

7.1.2 const的安全性

const的作用不限于在常数表达式里代替 #defines。如果用运行期间产生的值初始化一个变量而且知道在那个变量寿命期内它是不变的，用 const限定该变量，程序设计中这是一个很好的做法。如果偶然改变了它，编译器会给出一个出错信息。下面是一个例子：

```
//: SAFECONS.CPP -- Using const for safety
#include <iostream.h>

const int i = 100;  // Typical constant
const int j = i + 10; // Value from const expr
long address = (long)&j; // Forces storage
char buf[j + 10]; // Still a const expression

main() {
  cout << "type a character & CR:";
  const char c = cin.get(); // Can't change
  const char c2 = c + 'a';
  cout << c2;
  // ...
}
```

我们会发现，i是一个编译期间的常量，但j是从i中计算出来的。然而，由于i是一个常量，j的计算值来自一个常数表达式，而它自身也是一个编译期间的常量。紧接下面的一行需要 j的地址，所以迫使编译器给j分配存储空间。即使分配了存储空间，把j值保存在程序的某个地方，由于编译器知道j是常量，而且知道j值是有效的，所以，这仍不会妨碍在决定数组 buf的大小时使用j。

在主函数main()里，对于标识符c中有另一种const，因为其值在编译期间是不知道的。这意味着需要存储空间，而编译器不想在符号表里保留任何东西（和 C的方式一样）。初始化必须发生在定义的地方，而且一旦初始化，其值不能改变。我们看到 c2由c的值计算出来，也会看到这类常量的作用域与其他任何类型常量的作用域是一样的——这是对#define用法的另一种

改进。

实际上，如果想一个值不变，就应该使之成为常量（ const)。这不仅为防止意外的更改提供安全措施，也消除了存储和读内存操作，使编译器产生的代码更有效。

7.1.3 集合

const可以用于集合，但编译器不能把一个集合存放在它的符号表里，所以必须分配内存。在这种情况下，const意味着“ 不能改变的一块存储 ”。然而，其值在编译时不能被使用，因为编译器在编译时不需要知道存储的内容。这样，我们不能写：

```
//: CONSTAG.CPP -- Constants and aggregates

const int i[] = { 1, 2, 3, 4 };

//! float f[i[3]]; // Illegal

struct s { int i, j; };

const s S[] = { { 1, 2 }, { 3, 4 } };

//! double d[S[1].j]; // Illegal

main() {}
```

在一个数组定义里，编译器必须能产生这样的移动存储数组的栈指针代码。在上面这两种非法定义里，编译器给出“ 提示 ”是因为它不能在数组定义里找到一个常数表达式。

7.1.4 与C语言的区别

常量引进是在早期的C++版本中，当时标准C规范正在制订。那时，常量被看作是一个好的思想而被包含在C中。但是，C中的const意思是“ 一个不能被改变的普通变量 ”，在C中，它总是占用存储而且它的名字是全局符。C编译器不能把const看成一个编译期间的常量。在C中，如果写：

```
const bufsize=100；
char buf[bufsize]；
```

尽管看起来好像做了一件合理的事，但这将得到一个错误结果。因为 bufsize占用存储的某个地方，所以C编译器不知道它在编译时的值。在C语言中可以选择这样书写：

```
const bufsize；
```

这样写在C++中是不对的，而C编译器则把它作为一个声明，这个声明指明在别的地方有存储分配。因为C默认const是外部连接的，C++默认const是内部连接的，这样，如果在C++中想完成与C中同样的事情，必须用extern把连接改成外部连接：

```
extern const bufsize;//declaration only
```

这种方法也可用在C语言中。

在C语言中使用限定符const不是很有用，即使是在常数表达式里（必须在编译期间被求出）想使用一个已命名的值，使用const也不是很有用的。C迫使程序员在预处理器里使用#define。

7.2 指针

我们还可以使指针成为const指针。当处理const指针时，编译器仍将努力阻止存储分配并进行常量折叠，但在这种情况下，这些特征似乎很少有用。更重要的是，如果程序员以后想在程序代码中改变这种指针的使用，编译器将给出通知。这大大增加了安全性。

当使用带有指针的const时，有两种选择：或者const修饰指针正指向对象，或者const修饰存储在指针本身的地址里。这些语法在开始时有点使人混淆，但练习之后就好了。

7.2.1 指向const的指针

使用指针定义的技巧，正如任何复杂的定义一样，是在标识符的开始处读它并从里向外读。const指定那个“最靠近”的。这样，如果要使正指向的元素不发生改变，我们得写一个像这样的定义：

```
const int* x；
```

从标识符开始，是这样读的：“x是一个指针，它指向一个const int。”这里不需要初始化，因为说x可以指向任何东西（那是说，它不是一个const），但它所指的东西是不能被改变的。

这是一个容易混淆的部分。有人可能认为：要想指针本身不变，即包含在指针 x里的地址不变，可简单地像这样把const从一边移向另一边：

```
int const* x；
```

并非所有的人都很肯定地认为：应该读成“ x是一个指向int的const指针”。然而，实际上应读成“x是一个指向恰好是const的int普通指针”。即const又把它自己与int结合在一起，结果与前面定义一样。两个定义是一样的，这一点容易使人混淆。为使程序更具有可读性，我们应该坚持用第一种形式。

7.2.2 const指针

使指针本身成为一个const指针，必须把const标明的部分放在*的右边，如：

```
int d=1；
int* const x=&d；
```

现在它读成“ x是一个指针，这个指针是指向int的const指针”。因为现在指针本身是const指针，编译器要求给它一个初始化值，这个值在指针寿命期间不变。然而要改变它所指向的值是可以的，可以写*x=2；

也可以使用下面两种合法形式中的任何一种形式把一个const指针变为一个const对象：

```
int d=1；
const int* const x=&d； // (1)
int const* const x2=&d； // (2)
```

现在，指针和对象都不能改变。

一些人认为第二种形式更好。因为const总是放在被修改者的右边。但对于特定的代码类型来讲，程序员得自己决定哪一种形式更清楚。

• 格式

这本书主张，不管何时在一行里仅放一个指针定义，且在定义的地方初始化每个指针。正因为这一点，才可以把‘ * ’“ 附于 ”数据类型上：

```
int* u=&w；
```

int*本身好像是离散型的。这使代码更容易懂，可惜的是，事情并非如此。事实上，‘ * ’与标识符结合，而不是与类型结合。它可以被放在类型名和标识符之间的任何地方。所以，可以这样做：

```
int* u=&w, v=0;
```

它建立一个int* u和一个非指针int v。由于读者时常混淆这一点，所以最好用本书里所用的表示形式（即一行里只定义一个指针）。

7.2.3 赋值和类型检查

C++关于类型检查有其特别之处，这一点也扩展到指针赋值。我们可以把一个非 const对象的地址赋给一个const指针，因为也许有时不想改变某些可以改变的东西。然而，不能把一个const对象的地址赋给一个非 const指针，因为这样做可能通过被赋值指针改变这个 const指针。当然，总能用类型转换强制进行这样的赋值，但是，这不是一个好的程序设计习惯，因为这样就打破了对象的const属性以及由const提供的安全性。例如：

```
int d = 1;
const int e = 2;
int* u = &d; // OK -- d not const
int* v = &e; // illegal -- e const
int* w = (int*)&e; // legal but bad practice
```

虽然C++有助于防止错误发生，但如果程序员自己打破了这种安全机制，它也是无能为力的。

• 串字面值

限定词const是很严格的，const没被强调的地方是有关串字面值。也许有人写：

```
char* cp="howdy";
```

编译器将接受它而不报告错误。从技术上讲，这是一个错误，因为串字面值（这里是“ howdy ”）是被编译器作为一个常量串建立的，所引用串的结果是它在内存里的首地址。

所以串字面值实际上是常量串。然而，编译器把它们作为非常量看待，这是因为有许多现有的C代码是这样做的。当然，改变串字面值的做法还未被定义，虽然可能在很多机器上是这样做的。

7.3 函数参数和返回值

用const限定函数参数及返回值是常量概念另一个容易被混淆的地方。如果以值传递对象时，对用户来讲，用const限定没有意义（它意味着传递的参数在函数里是不能被修改的）。如果以常量返回用户定义类型的一个对象的值，这意味着返回值不能被修改。如果传递并返回地址，const将保证该地址内容不会被改变。

7.3.1 传递const值

如果函数是以值传递的，可用const限定函数参数，如：

```
void f1(const int i) {
  i++; // illegal -- compile-time error
}
```

这是什么意思呢？这是作了一个约定：变量初值不会被函数x()改变。然而，由于参数是以值传递的，因此要立即制作原变量的副本，这个约定对用户来说是隐藏的。

在函数里，const有这样的意义：参数不能被改变。所以它其实是函数创建者的工具，而不是函数调用者的工具。

为了不使调用者混淆，在函数内部用const限定参数优于在参数表里用const限定参数。可以用一个指针这样做，但更好的语法形式是“引用”，这是第10章讨论的主题。简言之，引用像一个被自动逆向引用的常指针，它的作用是成为对象的别名。为建立一个引用，在定义里使用&。所以，不引起混淆的函数定义看来像这样的：

```
void f2(int ic) {
  const int& i = ic;
  i++;  // illegal -- compile-time error
}
```

这又会得到一个错误信息，但这时对象的常量性（const）不是函数特征标志的部分；它仅对函数实现有意义，所以它对用户来说是不可见的。

7.3.2 返回const值

对返回值来讲，存在一个类似的道理，即如果从一个函数中返回值，这个值作为一个常量：

```
const int g()；
```

约定了函数框架里的原变量不会被修改。正如前面讲的，返回这个变量的值，因为这个变量被制成副本，所以初值不会被修改。

首先，这使const看起来没有什么意义。可以从这个例子中看到：返回常量值明显失去意义：

```
//: CONSTVAL.CPP -- Returning consts by value
// Has no meaning for built-in types

int f3() { return 1; }
const int f4() { return 1; }

main() {
  const int j = f3(); // Works fine
  int k = f4(); // But this works fine too!
}
```

对于内部数据类型来说，返回值是否是常量并没有关系，所以返回一个内部数据类型的值时，应该去掉const从而使用户程序员不混淆。

处理用户定义的类型时，返回值为常量是很重要的。如果一个函数返回一个类对象的值，其值是常量，那么这个函数的返回值不能是一个左值（即它不能被赋值，也不能被修改）。例如：

```
//: CONSTRET.CPP -- Constant return by value
// Result cannot be used as an lvalue
class X {
```

```
  int i;
public:
  X(int I = 0) : i(I) {}
  void modify() { i++; }
};

X f5() {
  return X();
}

const X f6() {
  return X();
}

void f7(X& x) { // Pass by non-const reference
  x.modify();
}

main() {
  f5() = X(1); // OK -- non-const return value
  f5().modify(); // OK
  f7(f5()); // OK
  // Causes compile-time errors:
//!  f6() = X(1);
//!  f6().modify();
//!  f7(f6());
}
```

f5()返回一个非const X对象，然而f6()返回一个const X对象。只有非const返回值能作为一个左值使用。换句话说，如果不让对象的返回值作为一个左值使用，当返回一个对象的值时，应该使用const。

返回一个内部数据类型的值时，const没有意义的原因是：编译器已经不让它成为一个左值（因为它总是一个值而不是一个变量）。仅当返回用户定义的类型对象的值时，才会出现上述问题。

函数f7()把它的参数作为一个非const引用（C++中另一种处理地址的办法，这是第10章讨论的主题）。从效果上讲，这与取一个非const指针一样，只是语法不同。

• 临时变量

有时候，在求表达式值期间，编译器必须建立临时对象。像其他任何对象一样，它们需要存储空间而且必须被构造和删除。区别是我们从来看不到它们——编译器负责决定它们的去留以及它们存在的细节。这里有一个关于临时变量的情况：它们自动地成为常量。因为我们通常接触不到临时对象，不能使用与之相关的信息，所以告诉临时对象做一些改变几乎肯定会出错。当程序员犯那样的错误时，由于使所有的临时变量自动地成为常量，编译器会向他发出错误警告。

类对象常量是怎样保存起来的，将在这一章的后面介绍。

7.3.3 传递和返回地址

如果传递或返回一个指针（或一个引用），用户取指针并修改初值是可能的。如果使这个指针成为常（const）指针，就会阻止这类事的发生，这是非常重要的。事实上，无论什么时候传递一个地址给一个函数，我们都应该尽可能用 const修饰它，如果不这样做，就使得带有指向const的指针函数不具备可用性。

是否选择返回一个指向const的指针，取决于我们想让用户用它干什么。下面这个例子表明了如何使用const指针作为函数参数和返回值：

```
//: CONSTP.CPP -- Constant pointer arg/return

void t(int*) {}

void u(const int* cip) {
//!  *cip = 2; // Illegal -- modifies value
  int i = *cip; // OK -- copies value
//!  int* ip2 = cip; // Illegal: non-const
}

const char* v() {
  // Returns address of static string:
  return "result of function v()";
}

const int* const w() {
  static int i;
  return &i;
}

main() {
  int x = 0;
  int* ip = &x;
  const int* cip = &x;
  t(ip);  // OK
//!  t(cip); // Not OK
  u(ip);  // OK
  u(cip); // Also OK
//!  char* cp = v(); // Not OK
  const char* ccp = v(); // OK
//!  int* ip2 = w(); // Not OK
  const int* const ccip = w(); // OK
  const int* cip2 = w(); // OK
//!  *w() = 1; // Not OK
}
```

函数t()把一个普通的非const指针作为一个参数，而函数u()把一个const指针作为参数。在

函数u()里，我们会看到试图修改const指针的内容是非法的。当然，我们可以把信息拷进一个非const变量。编译器也不允许使用存储在const指针里的地址来建立一个非const指针。

函数v()和w()测试返回的语义值。函数v()返回一个从串字面值中建立的const char*。在编译器建立了它并把它存储在静态存储区之后，这个声明实际上产生串字面值的地址。像前面提到的一样，从技术上讲，这个串是一个常量，这个常量由函数v()的返回值正确地表示。

w()的返回值要求这个指针及这个指针所指向的对象均为常量。像函数 v()一样，仅仅因为它是静态的，所以在函数返回后由 w()返回的值是有效的。函数不能返回指向局部栈变量的指针，这是因为在函数返回后它们是无效的，而且栈也被清理了。可返回的另一个普通指针是在堆中分配的存储地址，在函数返回后它仍然有效。

在函数main()中，函数被各种参数测试。函数t()将接受一个非const指针参数。但是，如果我们想传给它一个指向const的指针，那么就将不能防止t()丢下这个指针所指的内容不管，所以编译器会给出一个错误信息。函数 u()带一个const指针，所以它接受两种类型的参数。这样，带const指针函数比不带const指针函数更具一般性。

正如所期望的，函数v()返回值只可以被赋给一个const指针。编译器拒绝把函数w()的返回值赋给一个非const指针，而接受一个const int* const，但令人吃惊的是它也接受一个const int*，这与返回类型不匹配。正如前面所讲的，因为这个值（包含在指针中的地址）正被拷贝，所以自动保持这样的约定：原始变量不能被触动。因此，只有把 const int*const中的第二个const当作一个左值使用时（编译器会阻止这种情况），它才能显示其意义。

• 标准参数传递

在C语言中，值传递是很普通的，但是当我们想传递地址时，只能使用指针。然而，在C++中却不使用这两种方法。在C++中，传递一个参数时，先选择通过引用传递，而且是通过常量（const）引用。对程序员来说，这样做的语法与值传递是一样的，所以在指针方面没有混淆之处的——他们甚至不必考虑这个问题。对于类的创建者来说，传递地址总比传递整个类对象更有效，如通过常量（ const）引用来传递，这意味着函数将不改变该地址所指的内容，从用户程序员的观点来看，效果恰好与值传递一样。

由于引用的语法（看起来像值传递）的原因，传递一个临时对象给带有一个引用的函数，是可能的，但不能传递一个临时对象给带有一个指针的函数——因为它必须清楚地带有地址。所以，通过引用传递会产生一个在C中不会出现的新情形：一个总是常量的临时变量，它的地址可以被传递给一个函数。这就是为什么临时变量通过引用被传递给一个函数时，这个函数的参数一定是常量（const）引用。下面的例子说明了这一点：

```
//: CONSTTMP.CPP -- Temporaries are const

class X {};

X f() { return X(); } // Return by value

void g1(X&) {} // Pass by non-const reference
void g2(const X&) {} // Pass by const reference
main() {
  // Error: const temporary created by f():
//!  g1(f());
  // OK: g2 takes a const reference:
  g2(f());
}
```

函数f()返回类X的一个对象的值。这意味着立即取 f()的返回值并把它传递给其他函数时（正如g1()和g2()函数的调用），建立了一个临时变量，那个临时变量是常量。这样，函数g1()中的调用是错误的，因为g1()不带一个常量（const）引用，但是函数g2()中的调用是正确的。

7.4 类

这一部分介绍了const用于类的两种办法。程序员可能想在一个类里建立一个局部常量，将它用在常数表达式里，这个常数表达式在编译期间被求值。然而， const的意思在类里是不同的，所以必须使用另一技术——枚举，以达到同样的效果。

我们还可以建立一个类对象常量（const）（正如我们刚刚看到的，编译器总是建立临时类对象常量）。但是，要保持类对象为常量却比较复杂。编译器能保证一个内部数据类型为常量，但不能控制一个类中错综复杂的事物。为了保证一个类对象为常量，引进了 const成员函数：对于一个常量对象，只能调用const成员函数。

7.4.1 类里的const和enum

常数表达式使用常量的情况之一是在类里。典型的例子是在一个类里建立一个数组，并用const代替#define建立数组大小以及用于有关数组的计算。并把数组大小一直隐藏在类里，这样，如果用size表示数组大小，就可以把 size这个名字用在另一个类里而不发生冲突。然而预处理器从这些#define被定义的时起就把它们看成全程的，所以如用 #define就不会得到预期的效果。

起初读者可能认为合乎逻辑的选择是把一个const放在类里。但这不会产生预期的结果。在一个类里， const恢复它在C中的一部分意思。它在每个类对象里分配存储并代表一个值，这个值一旦被初始化以后就不能改变。在一个类里使用 const的意思是“在这个对象寿命期内，这是一个常量”。然而，对这个常量来讲，每个不同的对象可以含一个不同的值。

这样，在一个类里建立一个const时，不能给它初值。这个初始化工作必须发生在构造函数里，并且，要在构造函数的某个特别的地方。因为 const必须在建立它的地方被初始化，所以在构造函数的主体里， const必须已初始化了，否则，就只有等待，直到在构造函数主体以后的某个地方给它初始化，这意味着过一会儿才给 const初始化。当然，无法防止在在构造函数主体的不同地方改变const的值。

1. 构造函数初始化表达式表

构造函数有个特殊的初始化方法，称为构造函数初始化表达式表，起初用在继承里（继承是以后章节中有关面向对象的主题）。构造函数初始化表达式表——顾名思义，是出现在构造函数的定义里的——是一个出现在函数参数表和冒号后，但在构造函数主体开头的花括号前的“函数调用表”。这提醒人们，表里的初始化发生在构造函数的任何代码执行之前。这是把所有的const初始化的地方，所以类里的const正确形式是：

```
class fred {
  const size;
public:
  fred();
};

fred::fred() : size(100) {}
```

开始时，上面显示的构造函数初始化表达式表的形式容易使人们混淆，因为人们不习惯看到一个内部数据类型有一个构造函数。

2. 内部数据类型“构造函数”

随着语言的发展和人们为使用户定义类型像内部数据类型所作的努力，有时似乎使内部数据类型看起来像用户定义类型更好。在构造函数初始化表达式表里，可以把一个内部数据类型看成好像它有一个构造函数，就像下面这样：

```
class B {
  int i;
public:
  B(int I);
};

B::B(int I) : i(I) {}
```

这在初始化const数据成员时尤为典型，因为它们必须在进入函数体前被初始化。

我们还可以把这个内部数据类型的“构造函数”（仅指赋值）扩展为一般的情形，可以写：

```
float pi (3.14159);
```

把一个内部数据类型封装在一个类里以保证用构造函数初始化，是很有用的。例如，下面是一个integer类：

```
class integer {
  int i;
public:
  integer(int I = 0);
};

integer::integer(int I) : i(I) {}
```

现在，如果建立一个integer数组，它们都被自动初始化为零：

```
integer I[100];
```

与for循环和memset()相比，这种初始化不必付出更多的开销。很多编译器可以很容易地把它优化成一个很快的过程。

7.4.2 编译期间类里的常量

因为在类对象里进行了存储空间分配，编译器不能知道const的内容是什么，所以不能把它用作编译期间的常量。这意味着对于类里的常数表达式来说，const就像它在C中一样没有作用。我们不能这样写：

```
class bob {
  const size = 100;  // illegal
  int array[size];  // illegal
//...
```

在类里的const意思是“在这个特定对象的寿命期内，而不是对于整个类来说，这个值是不变的（const）”。那么怎样建立一个可以用在常数表达式里的类常量呢？一个普通的办法是使用一个不带实例的无标记的 enum。枚举的所有值必须在编译时建立，它对类来说是局部的，但常数表达式能得到它的值，这样，我们一般会看到：

```
class bunch {
  enum { size = 1000 };
  int i[size];
};
```

使用enum是不会占用对象中的存储空间的，枚举常量在编译时被全部求值。我们也可以明确地建立枚举常量的值：

```
enum { one=1,two=2,three};
```

对于整型enum，编译器从最后一个值继续计数，所以枚举常量three将取值3。

下面这个例子表明了在一个串指针栈里的enum的用法：

```
//: SSTACK.CPP -- Enums inside classes
#include <string.h>
#include <iostream.h>

class StringStack {
  enum { size = 100 };
  const char* stack[size];
  int index;
public:
   StringStack();
   void push(const char* s);
   const char* pop();
};

StringStack::StringStack() : index(0) {
  memset(stack, 0, size * sizeof(char*));
}

void StringStack::push(const char* s) {
  if(index < size)
    stack[index++] = s;
}
const char* StringStack::pop() {
  if(index > 0) {
    const char* rv = stack[--index];
    stack[index] = 0;
    return rv;
  }
  return 0;
}

const char* iceCream[] = {
  "pralines & cream",
  "fudge ripple",
  "jamocha almond fudge",
```

```
  "wild mountain blackberry",
  "raspberry sorbet",
  "lemon swirl",
  "rocky road",
  "deep chocolate fudge"
};

const ICsz = sizeof iceCream/sizeof *iceCream;

main() {
  StringStack SS;
  for(int i = 0; i < ICsz; i++)
    SS.push(iceCream[i]);
  const char* cp;
  while((cp = SS.pop()) != 0)
    cout << cp << endl;
}
```

注意push()带一个const char*参数，pop()返回一个const char*，stack保存const char*。如果不是这样，就不能用stringstack保存iceCream里的指针。然而，它不让程序员做任何事情以改变包含在Stringstack里的对象。当然，不是所有的串指针栈都有这个限制。

虽然会经常在以前的程序代码里看到使用enum技术，但C++还有一个静态常量static const，它在一个类里产生一个更灵活的编译期间的常量。这一点在将第9章描述。

• 枚举的类型检查

C中的枚举是相当原始的，只涉及整型值和名字，但不提供类型检查。在C++里，正如我们现在所期望的，类型概念是十分重要的，枚举正是这样要求的。我们建立了一个已命名的枚举时，我们就已经有效地建立了一个新的类型，就像一个类一样：在编译单元被翻译期间，枚举名字将成为一个保留字。

另外，C++中的枚举有一个比C中更严格的类型检查。假如我们有一个称为a的枚举类型color，就会注意这一点。在C中可以写a++，但在C++中不能这样写。这是因为枚举自增正在执行两个类型转换，其中一个类型在C++中是合法的，另一个是不合法的。首先，枚举的值隐蔽地从color转换到int，然后值增1，然后int又转回到color。在C++中，这样做是不允许的，因为color是一个与int不同的类型，无法知道blue加1恰好出现在颜色表里。如果要对color加1，那么它应该是一个类（有自增操作），而不是一个enum。不论什么时候写出了隐含对enum进行类型转换的代码，编译器都把它标记成危险的活动。

共用数据类型有类似的附加类型检查。

7.4.3 const对象和成员函数

可以用const限定类成员函数，这是什么意思呢？为了搞清楚这一点，必须首先掌握const对象的概念。

用户定义类型和内部数据类型一样，都可以定义一个const对象。例如：

```
const int i=1；
const blob B(2)；
```

这里，B是类型blob的一个const对象。它的构造函数被调用，且其参数为“ 2 ”。由于编译器强调对象为const的，因此它必须保证对象的数据成员在对象寿命期内不被改变。可以很容易地保证公有数据不被改变，但是怎么知道哪个成员函数会改变数据？又怎么知道哪个成员函数对于const对象来说是“ 安全 ”的呢？

如果声明一个成员函数为 const函数，则等于告诉编译器可以为一个 const对象调用这个函数。一个没有被特别声明为 const的成员函数被看成是将要修改对象中数据成员的函数，而且编译器不允许为一个const对象调用这个函数。

然而，不能就此为止。仅仅声明一个函数在类定义里是 const的，不能保证成员函数也如此定义，所以编译器迫使程序员在定义函数时要重申 const说明。(const已成为函数识别符的一部分，所以编译器和连接程序都要检查 const)。为确保函数的常量性，在函数定义中，如果我们改变对象中的任何成员或调用一个非 const成员函数，编译器都将发出一个出错信息，强调在函数定义期间函数被定义成 const函数。这样，可以保证声明为 const的任何成员函数能够按定义方式运行。

const放在函数声明前意味着返回值是常量，但这不合语法。必须把 const标识符放在参数表后。例如：

```
class X {
  int i;
public:
  int f() const;
};
```

关键字const必须用同样的方式重复出现在定义里，否则编译器把它看成一个不同的函数：

```
int X::f() const {return i ; }
```

如果f()试图用任何方式改变i或调用另一个非const成员函数，编译器把它标记成一个错误。

任何不修改成员数据的函数应该声明为const函数，这样它可以由const对象使用。

下面是一个比较const和非const成员函数的例子：

```
//: QUOTER.CPP -- Random quote selection
#include <iostream.h>
#include <stdlib.h> // Random number generator
#include <time.h> // To seed random generator

class quoter {
  int lastquote;
public:
  quoter();
  int Lastquote() const;
  const char* quote();
};

quoter::quoter(){
  lastquote = -1;
  time_t t;
  srand((unsigned) time(&t)); // Seed generator
}
```

```
int quoter::Lastquote() const {
  return lastquote;
}

const char* quoter::quote() {
  static const char* quotes[] = {
    "Are we having fun yet?",
    "Doctors always know best",
    "Is it ... Atomic?",
    "Fear is obscene",
    "There is no scientific evidence "
    "to support the idea "
    "that life is serious",
  };
  const qsize = sizeof quotes/sizeof *quotes;
  int qnum = rand() % qsize;
  while(lastquote >= 0 && qnum == lastquote)
    qnum = rand() % qsize;
  return quotes[lastquote = qnum];
}

main() {
  quoter q;
  const quoter cq;
  cq.Lastquote(); // OK
//!  cq.quote(); // Not OK; non const function
  for(int i = 0; i < 20; i++)
    cout << q.quote() << endl;
}
```

构造函数和析构函数都不是const成员函数，因为它们在初始化和清理时，总是对对象作些修改。quote()成员函数也不能是const函数，因为它在返回说明里修改数据成员lastquote。然而Lastquote()没做修改，所以它可以成为const函数，而且也可以被const对象cq安全地调用。

• 按位和与按成员 const

如果我们想要建立一个const成员函数，但仍然想在对象里改变某些数据，这时该怎么办呢？这关系到按位const和按成员const的区别。按位const意思是对象中的每个位是固定的，所以对象的每个位映像从不改变。按成员 const意思是，虽然整个对象从概念上讲是不变的，但是某个成员可能有变化。当编译器被告知一个对象是 const对象时，它将保护这个对象。这里我们要介绍在const成员函数里改变数据成员的两种方法。

第一种方法已成为过去，称为“强制转换const”。它以相当奇怪的方式执行。取this（这个关键字产生当前对象的地址）并把它强制转换成指向当前类型对象的指针。看来 this已经是我们所需的指针，但它是一个const指针，所以，还应把它强制转换成一个普通指针，这样就可以在运算中去掉常量性。下面是一个例子：

```
//: CASTAWAY.CPP -- "Casting away" constness

class Y {
  int i, j;
public:
  Y() { i = j = 0; }
  void f() const;
};

void Y::f() const {
//!   i++; // Error -- const member function
    ((Y*)this)->j++; //OK: cast away const-ness
}

main() {
  const Y yy;
  yy.f(); // Actually changes it!
}
```

这种方法可行，在过去的程序代码里可以看到这种用法，但这不是首选的技术。问题是：this没有用const修饰，这在一个对象的成员函数里被隐藏，这样，如果用户不能见到源代码（并找到用这种方法的地方），就不知道发生了什么。为解决所有这些问题，应该在类声明里使用关键字mutable，以指定一个特定的数据成员可以在一个const对象里被改变。

```
//: MUTABLE.CPP -- The "mutable" keyword

class Y {
  int i;
  mutable int j;
public:
  Y() { i = j = 0; }
  void f() const;
};

void Y::f() const {
//! i++; // Error -- const member function
    j++; // OK: mutable
}

main() {
  const Y yy;
  yy.f(); // Actually changes it!
}
```

现在类用户可从声明里看到哪个成员能够在一个const成员函数里被修改。

7.4.4 只读存储能力

如果一个对象被定义成const对象，它就成为被放进只读存储器（ROM）中的一个候选，

这经常是嵌入式程序设计中要考虑的重要事情。然而，只建立一个 const对象是不够的——只读存储能力的条件非常严格。当然，这个对象还应是按位 const的，而不是按成员const的。如果只通过关键字mutable实现按成员常量化的话，就容易看出这一点。如果在一个 const成员函数里的const被强制转换了，编译器可能检测不到这个。另外，

1) class或struct必须没有用户定义的构造函数或析构函数。

2) 这里不能有基类（将在关于继承的章节里谈到），也不能有包含用户定义的构造函数或析构函数的成员对象。

在只读存储能力类型的const对象中的任何部分上，有关写操做的影响没有定义。虽然适当形式的对象可被放进ROM里，但是目前还没有什么对象需要放进ROM里。

7.5 可变的（volatile）

volatile的语法与const是一样的，但是volatile的意思是“在编译器认识的范围外，这个数据可以被改变”。不知何故，环境正在改变数据（可能通过多任务处理），所以，volatile告诉编译器不要擅自做出有关数据的任何假定——在优化期间这是特别重要的。如果编译器说：“我已经把数据读进寄存器，而且再没有与寄存器接触”。一般情况下，它不需要再读这个数据。但是，如果数据是volatile修饰的，编译器不能作出这样的假定，因为可能被其他进程改变了，它必须重读这个数据而不是优化这个代码。

就像建立const对象一样，程序员也可以建立volatile对象，甚至还可以建立const volatile对象，这个对象不能被程序员改变，但可通过外面的工具改变。下面是一个例子，它代表一个类，这个类涉及到硬件通信：

```
//: VOLATILE.CPP -- The volatile keyword
class comm {
  const volatile unsigned char byte;
  volatile unsigned char flag;
  enum { bufsize = 100 };
  unsigned char buf[bufsize];
  int index;
public:
  comm();
  void isr() volatile;
  char read(int Index) const;
};

comm::comm() : index(0), byte(0), flag(0) {}

// Only a demo; won't actually work
// As an interrupt service routine:
void comm::isr() volatile {
  if(flag) flag = 0;
  buf[index++] = byte;
  // Wrap to beginning of buffer:
  if(index >= bufsize) index = 0;
}
```

```
char comm::read(int Index) const {
  if(Index < 0 || Index >= bufsize)
    return 0;
  return buf[Index];
}

main() {
  volatile comm Port;
  Port.isr(); // OK
//!  Port.read(0); // Not OK;
                 // read() not volatile
}
```

就像const一样，我们可以对数据成员、成员函数和对象本身使用 volatile，可以并且也只能为volatile对象调用volatile成员函数。

函数isr()不能像中断服务程序那样使用的原因是：在一个成员函数里，当前对象（ this）的地址必须被秘密地传递，而中断服务程序 ISR一般根本不要参数。为解决这个问题，可以使isr()成为静态成员函数，这是下面章节讨论的主题。

volatile的语法与const是一样的，所以经常把它们俩放在一起讨论。为表示可以选择两个中的任何一个，它们俩通称为c-v限定词。

7.6 小结

关键字const能将对象、函数参数、返回值及成员函数定义为常量，并能消除预处理器的值替代而不对预处理器有任何影响。所有这些都为程序设计提供了非常好的类型检查形式以及安全性。使用所谓的const correctness（在可能的任何地方使用const）已成为项目的救星。

对于忽视const而继续使用老的C代码的程序员， 第10、11两章将改变他们的做法，在那里将开始大量使用引用，那时将看到对函数参数使用 const是多么关键。

7.7 练习

1. 建立一个具有成员函数 fly()的名为bird的类和一个不含 fly()的名为rock的类。建立一个rock对象，取它的地址，把它赋给一个 void*。现在取这个void*，把它赋给一个bird*，通过那个指针调用函数fly()。C语言允许公开地通过void*赋值是C语言中的一个“ 缺陷 ”，为什么呢？您知道吗？

2. 建立一个包含const成员的类，在构造函数初始化表达式表里初始化这个 const成员，建立一个无标记的枚举，用它决定一个数组的大小。

3. 建立一个类，该类具有 const和非const成员函数。建立这个类的const和非const对象，试着为不同类型的对象调用不同类型的成员函数。

4. 创建一个函数，这个函数带有一个常量值参数。然后试着在函数体内改变这个参数。

5. 请自行证明C和C++编译器对于const的处理是不同的。创建一个全局的 const并将它用于一个常量表达式中；然后在C和C++下编译它。

第8章　内 联 函 数

C++继承C的一个重要特性是效率。假如 C++的效率显著地比 C低，程序设计者不会使用它。

在C中，保护效率的一个方法是使用宏 (macro)。宏可以不用普通函数调用就使之看起来像函数调用。宏的实现是用预处理器而不是编译器。预处理器直接用宏代码代替宏调用，所以就没有了参数压栈、生成汇编语言的 CALL、返回参数、执行汇编语言的 RETURN的时间花费。所有的工作由预处理器完成，因此，不用花费什么就具有了程序调用的便利和可读性。

C++中，使用预处理器宏存在两个问题。第一个问题在 C中也存在：宏看起来像一个函数调用，但并不总是这样。这就隐藏了难以发现的错误。第二个问题是 C++特有的：预处理器不容许存取私有(private)数据。这意味着预处理器宏在用作成员函数时变得非常无用。

为了既保持预处理器宏的效率又增加安全性，而且还能像一般成员函数一样可以在类里访问自如，C++用了内联函数(inline function)。本章我们将研究C++预处理器宏存在的问题、C++中如何用内联函数解决这些问题以及使用内联函数的方针。

8.1　预处理器的缺陷

预处理器宏存在的关键问题是我们可能认为预处理器的行为和编译器的行为一样。当然，有意使宏在外观上和行为上与函数调用一样，因此容易被混淆。当微妙的差异出现时，问题就出现了。

考虑下面这个简单例子：

```
#define f (x) (x+1)
```

现在假如有一个像下面的f的调用

```
f (1)
```

预处理器展开它，出现下面不希望的情况：

```
(x)  (x+1) (1)
```

出现这个问题是因为在宏定义中 f和括号之间存在空格缝隙。当定义中的这个空格取消后，实际上调用宏时可以有空格空隙。像下面的调用：

```
f   (1)
```

依然可以正确地展开为：

```
(1 + 1)
```

上面的例子虽然微不足道但问题非常明显。在宏调用中使用表达式作为参数时，问题就出现了。

存在两个问题。第一个问题是表达式在宏内展开，所以它们的优先级不同于我们所期望的优先级。例如：

```
#define floor(x,b) x>=b?0:1
```

现在假如对参数使用表达式

```
if(floor(a&0x0f,0x07)) // ...
```

宏将展开成：

```
if(a&0x0f>=0x07?0:1)
```

因为&的优先级比>=的低，所以宏的展开结果将会使我们惊讶。一旦发现这个问题，可以通过在宏定义内使用括弧来解决。上面的定义可改写如下：

```
#define floor(x,b) ((x)>=(b)?0:1)
```

发现问题可能很难，我们可能一直认为宏的行为是正确的。在前面没有加括号的版本的例子中，大多数表达式将正确工作，因为 >=的优先级比像+、/、--甚至位移动操作符的优先级都低。因此，很容易想到它对于所有的表达式都正确，包括那些位逻辑操作符。

前面的问题可以通过谨慎地编程来解决：在宏中将所有的内容都用括号括起来。第二个问题则更加微妙。不像普通函数，每次在宏中使用一个参数，都对这个参数求值。只要宏仅用普通变量调用，这个求值就开始了。但假如参数求值有副作用，那么结果可能出乎预料，并肯定不能模仿函数行为。

例如，下面这个宏决定它的参数是否在一定范围：

```
#define band(x) (((x)>5 && (x)<10) ? (x) : 0)
```

只要使用一个“普通”参数，宏和真的函数工作得非常相像。但只要我们松懈并开始相信它是一个真的函数时，问题就开始出现了。

```
//: MACRO.CPP -- Side effects with macros
#include <fstream.h>
#define band(x) (((x)>5 && (x)<10) ? (x) : 0)

main() {
  ofstream out("macro.out");
  for(int i = 4; i < 11; i++) {
    int a = i;
    out << "a = " << a << endl << '\t';
    out << "band(++a)=" << band(++a) << endl;
    out << "\t a = " << a << endl;
  }
}
```

下面是这个程序的输出，它完全不是我们想从真正的函数期望得到的结果：

```
a = 4
       band(++a)=0
        a = 5
a = 5
       band(++a)=8
        a = 8
a = 6
       band(++a)=9
        a = 9
a = 7
       band(++a)=10
```

```
        a = 10
a = 8
      band(++a)=0
        a = 10
a = 9
      band(++a)=0
        a = 11
a = 10
      band(++a)=0
        a = 12
```

当a等于4时，测试了条件表达式第一部分，但它不满足条件，而表达式只求值一次，所以宏调用的副作用是a等于5，这是在相同的情况下普通函数调用得到的结果。但当数字在范围之内时，两个表达式都测试，产生两次自增操作。产生这个结果是由于再次对参数操作。一旦数字出了范围，两个条件仍然测试，所以也产生两次自增操作。根据参数不同产生的副作用也不同。

很清楚，这不是我们想从看起来像函数调用的宏中所希望的。在这种情况下，明显有效的解决方法是设计真正的函数。当然，如果多次调用函数将会增加额外的开销并可能降低效率。不幸的是，问题可能并不总是如此明显。我们可能不知不觉地得到一个包含混合函数和宏在一起的库函数，所以像这样的问题可能隐藏了一些难以发现的缺陷。例如，在 STDIO.H 中的putc( )宏可能对它的第二个参数求值两次。这在标准 C中作了详细说明。宏toupper( )不谨慎地执行也会对第二个参数求值超过两次。如在使用 toupper(*p++) [1] 时就会产生不希望的结果。

宏和访问

当然，对于C需要对预处理器宏谨慎地编码和使用。即使不是因为宏不是成员函数所需要的范围概念这一原因，我们也会在C++中避免使用它所带来的麻烦。预处理器简单地执行原文替代，所以不可能用下面这样或近似的形式写：

```
class X {
  int i;
public:
#define val (X::i)  //Error
```

另外，这里没有指明我们正在涉及哪个对象。在宏里简直没有办法表示类的范围。没有能取代预处理器宏的方法，程序设计者出于效率考虑，不得不让一些数据成员成为 public类型，这样就会暴露内部的实现并妨碍在这个实现中的改变。

8.2 内联函数

在解决C++中宏存取私有的类成员的问题过程中，所有和预处理器宏有关的问题也随着消失了。这是通过使宏被编译器控制来实现的。在 C++中，宏的概念是作为内联函数来实现的，而内联函数无论在任何意义上都是真正的函数。唯一不同之处是内联函数在适当时像宏一样展开，所以函数调用的开销被取消。因此，应该永远不使用宏，只使用内联函数。

[1] 在Andraw Koenig所著的书《C的陷阱和缺陷》（Addision-Wesley,1989）中将更详细地阐述。

任何在类中定义的函数自动地成为内联函数，但也可以使用 inline关键字放在类外定义的函数前面使之成为内联函数。但为了使之有效，必须使函数体和声明结合在一起，否则，编译器将它作为普通函数对待。因此

```
inline int PlusOne(int x);
```

没有任何效果，仅仅只是声明函数（这不一定能够在稍后某个时候得到一个内联定义）。成功的方法如下：

```
inline int PlusOne(int x) { return  ++x ;}
```

注意，编译器将检查函数参数列表使用是否正确，并返回值（进行必要的转换）。这些事情是预处理器无法完成的。假如对于上面的内联函数，我们写成一个预处理器宏的话，将有不想要的副作用。

一般应该把内联定义放在头文件里。当编译器看到这个定义时，它把函数类型（函数名 + 返回值）和函数体放到符号表里。当使用函数时，编译器检查以确保调用是正确的且返回值被正确使用，然后将函数调用替换为函数体，因而消除了开销。内联代码的确占用空间，但假如函数较小，这实际上比为了一个普通函数调用而产生的代码（参数压栈和执行 CALL）占用的空间还少。

在头文件里，内联函数默认为内部连接——即它是 static，并且只能在它被包含的编译单元看到。因而，只要它们不在相同的编译单元中声明，在内联函数和全局函数之间用同样的名字也不会在连接时产生冲突。

8.2.1 类内部的内联函数

为了定义内联函数，通常必须在函数定义前面放一个 inline关键字。但这在类内部定义内联函数时并不是必须的。任何在类内部定义的函数自动地为内联函数。如下例：

```
//: INLINE.CPP -- Inlines inside classes
#include <iostream.h>

class point {
  int i, j, k;
public:
  point() { i = j = k = 0; }
  point(int I, int J, int K) {
    i = I;
    j = J;
    k = K;
  }
  void print(const char* msg = "") const {
    if(*msg) cout << msg << endl;
    cout << "i = " << i << ", "
         << "j = " << j << ", "
         << "k = " << k << endl;
  }
};

main() {
```

```
  point p, q(1,2,3);
  p.print("value of p");
  q.print("value of q");
}
```

当然，因为类内部的内联函数节省了在外部定义成员函数的额外步骤，所以我们一定想在类声明内每一处都使用内联函数。但应记住，内联的目的是减少函数调用的开销。假如函数较大，那么花费在函数体内的时间相对于进出函数的时间的比例就会较大，所以收获会较小。而且内联一个大函数将会使该函数所有被调用的地方都做代码复制，结果代码膨胀而在速度方面获得的好处却很少或者没有。

8.2.2 存取函数

在类中内联函数的最重要的用处之一是用于一种叫存取函数的函数。这是一个小函数，它容许读或修改对象状态——即一个或几个内部变量。类内存取函数使用内联方式重要的原因在下面的例子中可以看到。

```
//: ACCESS.CPP -- Inline access functions

class access {
  int i;
public:
  int read() const { return i; }
  void set(int I) { i = I; }
};

main() {
  access A;
  A.set(100);
  int x = A.read();
}
```

这里，在类的设计者控制下，将类里面状态变量设计为私有 (private)，类的使用者就永远不会直接和它们发生联系了。对私有 (private)数据成员的所有存取只可以通过成员函数接口进行。而且，这种存取是相当有效的。例如对于函数 read( )。若没用内联函数，对 read( )调用产生的代码将包括对this压栈和执行汇编语言CALL。对于大多数机器，产生的代码将比内联函数产生的代码大一些，执行的时间肯定要长一些。

不用内联函数，考虑效率的类设计者将忍不住简单地使 i为公共(public)成员，从而通过让用户直接存取i 而节约开销。从设计的角度看，这是很不好的。因为 i将成为公共界面的一部分，所以意味着类设计者决不能修改它。我们将和称为 i的一个int类型变量打交道。这是一个问题，因为我们可能在稍后觉得用一个 float变量比用一个int 变量代表状态信息更有用一些，但因为 int i是公共接口的一部分，所以我们不能改变它。另一方面，假如我们总是使用成员函数读和修改一个对象的状态信息，那么就可以满意地修改对象内部一些描述（应该永远打消在编码和测试之前能使我们的设计完善的念头）。

• 存取器(accessors)和修改器(mutators)

一些人进一步把存取函数的概念分成存取器（从一个对象读状态信息）和修改器（修改状

态信息)。而且，可以用重载函数对存取器和修改器提供相同名字的函数，如何调用函数决定了我们是读还是修改状态信息。

```
//: RECTANGL.CPP -- Accessors & mutators

class rectangle {
  int Width, Height;
public:
  rectangle(int W = 0, int H = 0)
    : Width(W), Height(H) {}
  int width() const { return Width; } // Read
  void width(int W) { Width = W; } // Set
  int height() const { return Height; } // Read
  void height(int H) { Height = H; } // Set
};

main() {
  rectangle R(19, 47);
  // Change width & height:
  R.height(2 * R.width());
  R.width(2 * R.height());
}
```

构造函数使用构造函数初始表达式表（在第 7章中作了简介，在13中章将详细介绍）来初始化Width和Height值（对于内部数据类型使用伪编译器调用形式)。

当然，存取器和修改器对于一个内部变量不必只是简单的传递途径。有时，它们可以执行一些计算。下面的例子使用标准的C库函数中的时间函数来生成简单的Time类：

```
//: CPPTIME.H -- A simple time class
#ifndef CPPTIME_H_
#define CPPTIME_H_
#include <time.h>
#include <string.h>

class Time {
  time_t T;
  tm local;
  char Ascii[26];
unsigned char lflag, aflag;
void updateLocal() {
  if(!lflag) {
    local = *localtime(&T);
    lflag++;
  }
}
void updateAscii() {
  if(!aflag) {
```

```
      updateLocal();
      strcpy(Ascii, asctime(&local));
      aflag++;
    }
  }
public:
  Time() { mark(); }
  void mark() {
    lflag = aflag = 0;
    time(&T);
  }
  const char* ascii() {
    updateAscii();
    return Ascii;
  }
  // Difference in seconds:
  int delta(Time* dt) const {
    return difftime(T, dt->T);
  }
  int DaylightSavings() {
    updateLocal();
    return local.tm_isdst;
  }
  int DayOfYear() { // Since January 1
    updateLocal();
    return local.tm_yday;
  }
  int DayOfWeek() { // Since Sunday
  updateLocal();
  return local.tm_wday;
}
int Since1900() { // Years since 1900
  updateLocal();
  return local.tm_year;
}
int Month() { // Since January
  updateLocal();
  return local.tm_mon;
}
int DayOfMonth() {
  updateLocal();
  return local.tm_mday;
}
int Hour() { // Since midight, 24-hour clock
  updateLocal();
  return local.tm_hour;
```

```
  }
  int Minute() {
    updateLocal();
    return local.tm_min;
  }
  int Second() {
    updateLocal();
    return local.tm_sec;
  }
};
#endif // CPPTIME_H_
```

标准C库函数对于时间有多种表示，它们都是类Time的一部分。但全部更新它们是没有必要的，所以time_t T被用作基本的表示法，tm local 和ASCII字符表示法Ascii都有一个标记来显示它们是否已被更新为当前的时间time_t。两个私有函数updateLocal( )和 updateAscii( )检查标记，并有条件地执行更新。

构造函数调用mark( )函数时（用户也可以调用它，强迫对象表示当前时间）也就清除了两个标记，这时当地时间和ASCII表示法是无效的。函数ascii( )调用updateAscii( )， 因为函数ascii( )使用静态数据，假如它被调用，则这个静态数据被重写，所以updateAscaii( )把标准C库函数的结果拷贝到内部缓冲器里。返回值就是内部缓冲器的地址。

所有以DaylightSaving( )开始的函数都使用函数updateLocal( )，这就使得复合的内联函数变得相当大。这似乎不划算，尤其是考虑到可能不经常调用这些函数。但这不意味着所有的函数都应该用非内联函数。假如让updateLocal( )作为一个内联函数，它的代码将被复制在所有的非内联函数里，也能节省额外的开销。

下面是一个小的测试程序：

```
//: CPPTIME.CPP -- Testing a simple time class
#include "..\7\cpptime.h"
#include <iostream.h>

main() {
  Time start;
  for(int i = 1; i < 1000; i++) {
    cout << i << ' ';
    if(i%10 == 0) cout << endl;
  }
  Time end;
  cout << endl;
  cout << "start = " << start.ascii();
  cout << "end = " << end.ascii();
  cout << "delta = " << end.delta(&start);
}
```

在这个例子里，一个Time对象被创建，然后执行一些时延动作，接着创建第2个Time对象来标记结束时间。这些用于显示开始时间、结束时间和消耗的时间。

8.3 内联函数和编译器

为了理解内联何时有效，应该先理解编译器遇到一个内联函数时将做什么。对于任何函数，编译器在它的符号表里放入函数类型（即包括名字和参数类型的函数原型及函数的返回类型）。另外，编译器看到内联函数和内联函数的分析没有错误时，函数的代码也被放入符号表。代码是以源程序形式存放还是以编译过的汇编指令形式存放取决于编译器。

调用一个内联函数时，编译器首先确保调用正确，即所有的参数类型必须是正确类型或编译器必须能够将类型转换为正确类型，并且返回值在目标表达式里应该是正确类型或可改变为正确类型。当然，编译器对任何类型函数都是这样做的，这与预处理器显著不同，因为预处理器不能检查类型和进行转换。

假如所有的函数类型信息符合调用的上下文的话，内联函数代码就会直接替换函数调用，消除了调用的开销。假如内联函数也是成员函数，对象的地址（this)就会被放入合适的地方，这当然也是预处理器不能执行的。

8.3.1 局限性

这儿有两种编译器不能处理内联的情况。在这些情况下，它就像对非内联函数一样，通过定义内联函数和为函数建立存贮空间，简单地将其转换为函数的普通形式。假如它必须在多编译单元里做这些（通常将产生一个多定义错误），连接器就会被告知忽略多重定义。

假如函数太复杂，编译器将不能执行内联。这取决于特定编译器，但大多数编译器这时都会放弃内联方式，因为这时内联将可能不为我们提供任何效率。一般地，任何种类的循环都被认为太复杂而不扩展为内联函数。循环在函数里可能比调用要花费更多的时间。假如函数仅有一条简单语句，编译器可能没有任何内联的麻烦，但假如有许多语句，调用函数的开销将比执行函数体的开销少多了。记住，每次调用一个大的内联函数，整个函数体就被插入在函数调用的地方，所以没有任何引人注目的执行上的改进就使代码膨胀。本书的一些例子可能超过了一定的合理内联尺寸。

假如我们要显式或隐含地取函数地址，编译器也不能执行内联。因为这时编译器必须为函数代码分配内存从而为我们产生一个函数的地址。但当地址不需要时，编译器仍可能内联代码。

我们必须理解内联仅是编译器的一个建议，编译器不强迫内联任何代码。一个好的编译器将会内联小的、简单的函数，同时明智地忽略那些太复杂的内联。这将给我们想要的结果——具有宏效率的函数调用。

8.3.2 赋值顺序

假如我们想象编译器对执行内联做了些什么时，我们可能糊里糊涂地认为存在着比事实上更多的限制。特别是，假如一个内联函数对于一个还没有在类里声明的函数进行向前引用，编译器就可能不能处理它。

```
//: EVORDER.CPP -- Inline evaluation order

class forward {
  int i;
public:
  forward() : i(0) {}
```

```
  // Call to undeclared function:
  int f() const { return g() + 1; }
  int g() const { return i; }
};

main() {
  forward F;
  F.f();
}
```

虽然函数g( )还没有定义，但在函数f( )里对函数g( )进行了调用。这是可行的，因为语言定义规定非内联函数直到类声明结束才赋值。

当然，函数g( )也调用函数f( )，我们将得到一组递归调用，这些递归对于编译器进行内联是过于复杂了。（应该在函数f( )或g( )里也执行一些测试来强迫它们之一“停止”，否则递归将是无穷的）。

8.3.3 在构造函数和析构函数里隐藏行为

构造函数和析构函数是两个使我们易于认为内联比它实际上更有效的函数。构造函数和析构函数都可能隐藏行为，因为类可以包含子对象，子对象的构造函数和析构函数必须被调用。这些子对象可能是成员对象，或可能由于继承（继承还没有介绍）而存在。下面是一个有成员对象的例子。

```
//: HIDDEN.CPP -- Hidden activites in inlines
#include <iostream.h>

class member {
  int i, j, k;
public:
  member(int x = 0) { i = j = k = x; }
  ~member() { cout << "~member" << endl; }
};

class withMembers {
  member Q, R, S; // Have constructors
  int i;
public:
  withMembers(int I) : i(I) {} // Trivial?
  ~withMembers() {
    cout << "~withMembers" << endl;
  }
};

main() {
  withMembers WM(1);
}
```

在类withMembers里，内联的构造函数和析构函数看起来似乎很直接和简单，但其实很复杂。成员对象Q、P和S的构造函数和析构函数被自动调用，这些构造函数和析构函数也是内联的，所以它们和普通的成员函数的差别是显著的。这并不是意味着应该使构造函数和析构函数定义为非内联的。一般说来，快速地写代码来建立一个程序的初始“轮廓”时，使用内联函数经常是便利的。但假如要考虑效率，内联是值得注意的一个问题。

8.4 减少混乱

在本书里，类里放入内联定义的简单性、精练性是非常有用的，因为这样更容易放在一页或一屏中，看起来更方便一些。但Dan Saks指出，在一个真正的工程里，这将造成类接口混乱，因此使类难以使用。他用拉丁文 *in situ* 来表示定义在类里的成员函数(在适当的位置上)，并主张所有的定义都放在类外面以保持接口清楚。他认为这并不妨碍最优化。假如想优化，那么使用关键字inline。使用这个方法，前面（8.2.2节）RECTANGL.CPP例子修改如下：

```
//: NOINSITU.CPP -- Removing in situ functions

class rectangle {
  int Width, Height;
public:
  rectangle(int W = 0, int H = 0);
  int width() const; // Read
  void width(int W); // Set
  int height() const; // Read
  void height(int H); // Set
};

inline rectangle::rectangle(int W, int H)
  : Width(W), Height(H) {
}

inline int rectangle::width() const {
  return Width;
}

inline void rectangle::width(int W) {
  Width = W;
}

inline int rectangle::height() const {
  return Height;
}

inline void rectangle::height(int H) {
  Height = H;
}
```

```
main() {
  rectangle R(19, 47);
  // Transpose width & height:
  R.height(R.width());
  R.width(R.height());
}
```

现在假如想比较一下内联函数与非内联函数的效果，可以简单地移去关键字 inline。（内联函数通常应该放在头文件里，但非内联函数必须放在它们自己的编译单元里。）假如想把函数放入文件，只用简单的剪切和粘贴操作。*In situ*函数需要更多的操作，且更可能出错。这个方法的另外一个争论是我们可能总是对于函数定义使用一致的格式化类型，有些并没有总是在*in situ*函数中出现。

8.5 预处理器的特点

前面我说过，我们几乎总是希望使用内联函数代替预处理器宏。然而当在标准 C预处理器（通过继承也是C++预处理器）里使用3个特别的特征时却是例外：字符串定义、字符串串联和标志粘贴。字符串定义的完成是用 #指示，它容许设一个标识符并把它转化为字符串，然而字符串串联发生在当两个相邻的字符串没有分隔符时，在这种情况下字符串组合在一起。在写调试代码时，这两个特征是非常有效的。

```
#define DEBUG(X)  cout<<#X " = " << X << endl
```

上面的这个定义可以打印任何变量的值。我们也可以得到一个跟踪信息，在此信息里打印出它们执行的语句。

```
#define TRACE(S) cout << #S << endl; S
```

#S定义了要输出的语句。第2个S重申了语句，所以这个语句被执行。当然，这可能会产生问题，尤其是在一行for循环中。

```
for (int i = 0 ; i < 100 ; i++ )
  TRACE(f(i)) ;
```

因为在TRACE( )宏里实际上有两个语句，所以一行 for循环只执行第一个。解决方法是在宏中用逗号代替分号。

标志粘贴

标志粘贴在写代码时是非常有用的。它让我们设两个标识符并把它们粘贴在一起自动产生一个新的标识符。例如：

```
#define FIELD(A) char* A##_string; int A##_size
class record {
  FIELD(one);
  FIELD(two);
  FIELD(three);
  // ...
};
```

每次调用FIELD( )宏，将产生一个保存字符串的标识符和另一个保存字符串长度的标识符。

它不仅易读而且消除了编码出错，使维护更容易。但注意宏的名字中使用大写字母。这是对我们非常有帮助的习惯，因为它告诉读者这是宏而不是函数。所以假如存在问题，它可以作为一个提示。

8.6 改进的错误检查

为本书其余部分改进错误检查是很方便的。用内联函数可以简单地包括一个文件而不用担心连接什么。到目前为止，assert( )宏已用于“错误检查”，但它真正用处是调试并终将被能够在运行时提供更多有用信息的东西代替。何况异常处理程序（在17章介绍）已提供了更多的处理这些错误的有效的方法。

这是预处理器仍然有用的另一个例子，因为_FILE_和_LINE_指示仅和预处理器一起起作用并用在assert( )宏里。假如assert( )宏在一个错误函数里被调用，它仅打印出错函数的行号和文件名字而不是调用错误函数。这儿显示了使用宏联接（许多是 assert( )方法）函数的方法，紧接着调用assert( )（程序调试成功后这由一个#define NDEBUG消除）。

下面的头文件将放在书的根目录中，所以它可以从所有的章节里得到。“Allege”是assert的同义词。

```
//: ALLEGE.H -- Error checking
#ifndef ALLEGE_H_
#define ALLEGE_H_
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

inline void
allege_error(int val, const char * msg){
  if(!val) {
    fprintf(stderr, "error: %s\n", msg);
#ifdef NDEBUG
    exit(1);
#endif
  }
}

#define allege(expr, msg) \
{  allege_error((expr) ? 1 : 0, msg); \
  assert(expr); }

#define allegemem(expr) \
  allege(expr, "out of memory")

#define allegefile(expr) \
  allege(expr, "could not open file")
#endif // ALLEGE_H_
```

函数allege_error( )有两个参数：一个是整型表达式的值，另一个是这个值为 false时需打印

的消息。函数fprintf( )代替iostreams是因为在只有少量错误的情况下，它工作得更好。假如这不是为调试建立的，exit(1)被调用以终止程序。

allege( )宏使用三重if-then-else强迫计算表达式expr求值。在宏里调用了allege_error( )，接着是assert( )，所以我们能在调试时获得 assert( )的好处——因为有些环境紧密地把调试器和assert( )结合在一起。

allegefile( )宏和allegemen( )宏分别是allege( )宏用于检查文件和内存的专用版本。这个代码提供了出错报告的必要的最少信息，但我们可以在这个框架基础上增加它。

下面是测试ALLEGE.H简单例子。

```
//: ERRTEST.CPP -- Testing the allege() macro
// #define NDEBUG // turn off asserts
#include "..\allege.h"
#include <fstream.h>

main() {
  int i = 1;
  allege(i, "value must be nonzero");
  void* m = malloc(100);
  allegemem(m);
  ifstream nofile("nofile.xxx");
  allegefile(nofile);
}
```

去掉下面这行的注释符后，我们就知道这个程序是如何变为成品的：

```
//#define NDEBUG  // turn off asserts
```

对于本书其余部分，将一律用allege( )宏代替assert( )，只有个别只须在调试时检查而运行时不需的情况才用assert( )。

8.7 小结

能够隐藏类下面的实现是关键的，因为在以后我们有可能想修改那个实现。我们可能为了效率这样做，或因为对问题有了更好的理解，或因为有些新类变得可用而想在实现里使用这些新类。任何危害实现隐蔽性的东西都会减少语言的灵活性。这样，内联函数就显得非常重要，因为它实际上消除了预处理器宏和伴随它们的问题。通过用内联函数方式，成员函数可以和预处理器宏一样有效。

当然，内联函数也许会在类定义里被多次使用。因为它更简单，所以程序设计者都会这样做。但这不是大问题，因为以后期待程序规模减少时，可以将函数移出内联而不影响它们的功能。开发指南应该是“首先是使它起作用，然后优化它。”

8.8 练习

1. 将第7章练习2例子增加一个内联构造函数和一个称为Print( )的内联成员函数，这个函数用于打印所有数组的值。

2. 对于第3章的NESTFRND.CPP例子，用内联函数代替所有的成员函数，使它们成为非

*in situ*内联函数。同时再把initialize( )函数改成构造函数。

3. 使用第6章NL.CPP，在它自己的头文件里，将nl转变为内联函数。

4. 创建一个类A，具有能自我宣布的缺省构造函数。再写一个新类B，将A的一个对象作为B的成员，并为类B写一个内联构造函数。创建一个B对象的数组并看看发生了什么事。

5. 从练习4里创建大量的对象并使用Time类来计算非内联构造函数和内联构造函数之间的时间差别（假如我们有剖析器，也试着使用它。）

第9章 命名控制

创建名字是编程中最基本的活动，当一个项目中包含大量名字时，名字很容易冲突。 C++允许我们对名字的产生和名字的可见性进行控制，包括名字的存储位置以及名字的连接。

static这个关键字早在人们知道“重载”这个词之前就在C语言中被重载了，在C++中又增加了新的含义。static最基本的含义是指“位置不变的某个东西”（像“静电”），这里则指内存中的物理位置或文件中的可见性。

在这一章里，我们将看到static是怎样控制存储和可见的，还将看到一种通过C++的名字空间特征来控制访问名字的改进方法。我们还可以发现怎样使用C语言中编写并编译过的函数。

9.1 来自C语言中的静态成员

在C和C++中，static都有两种基本的含义，并且这两种含义经常是互相有冲突的：

1) 在固定的地址上分配，也就是说对象是在一个特殊的静态数据区上创建的，而不是每次函数调用时在堆栈上产生的。这也是静态存储的概念。

2) 对一个特定的编译单位来说是本地的（就像我们在后面将要看到的，这在C++中包括类的范围）。这里static控制名字的可见性，所以这个名字在这个单元或类之外是不可见的。这也描述了连接的概念，它决定连接器将看到哪些名字。

本节将着重讨论static的这两个含义，这些都是从C中继承来的。

9.1.1 函数内部的静态变量

通常，在函数体内定义一个变量时，编译器使得每次函数调用时堆栈的指针向下移一个适当的位置，为这些内部变量分配内存。如果这个变量有一个初始化表达式，那么每当程序运行到此处，初始化就被执行。

然而，有时想在两次函数调用之间保留一个变量的值，我们可以通过定义一个全局变量来实现这点，但这样一来，这个变量就不仅仅受这个函数的控制。 C和C++都允许在函数内部创建一个static对象，这个对象将存储在程序的静态数据区中，而不是在堆栈中。这个对象只在函数第一次调用时初始化一次，以后它将在两次函数之间保持它的值。比如，下面的函数每次调用时都返回一个字符串中的下一个字符。

```
//: STATFUN.CPP -- Static vars inside functions
#include <iostream.h>
#include "..\allege.h"

char onechar(const char* string = 0) {
  static const char* s;
  if(string) {
    s = string;
    return *s;
  }
```

```
  else
    allege(s, "un-initialized s");
  if(*s == '\0')
    return 0;
  return *s++;
}

char* a = "abcdefghijklmnopqrstuvwxyz";

main() {
  // Onechar(); // causes allege()
  onechar(a); // Initializes s to a
  char c;
  while((c = onechar()) != 0)
    cout << c << endl;
}
```

static char* s在每次onechar()调用时保留它的值，因为它存放在程序的静态数据区而不是存储在函数的堆栈中。当我们用一个字符指针作参数调用 onechar()时，参数值被赋给s，然后返回字符串的第一个字符。以后每次调用 onechar()都不用带参数，函数将使用缺省参数 0作为string的值，函数就会继续用以前初始化的 s值取字符，直到它到达字符串的结尾标志 —— 空字符为止。这时，字符指针就不会再增加了，这样，指针不会越过字符串的末尾。

但是，如果调用onechar()时没有参数而且s以前也没有初始化，那会怎样呢？我们也许会在s定义时提供一个初始值：

```
static char* s=0；
```

但如果说没有为一个预定义类型的静态变量提供一个初始值的话，编译器也会确保在程序开始时它被初始化为零（转化为适当的类型），所以在onechar()中，函数第一次调用时s将被赋值为零，这样if(!s)后面的程序就会被执行。

上例中s的初始化是很简单的，其实对一个静态对象的初始化（与其他对象的初始化一样）可以是任意的常量表达式，它可以包括常量及在此之前已声明过的变量和函数。

1. 函数体内部的静态对象

用户自定义的静态变量同一般的静态对象的规则是一样的，而且它同样也必须有初始化操作。但是，零赋值只对预定义类型有效，用户自定义类型必须用构造函数来初始化。因此，如果我们在定义一个静态对象时没有指定构造函数参数，这个类就必须有缺省的构造函数。请看下例：

```
//: FUNOBJ.CPP -- Static objects in functions
#include <iostream.h>

class X {
  int i;
public:
  X(int I = 0) : i(I) {} // Default
  ~X() { cout << "X::~X()" << endl; }
};
void f() {
  static X x1(47);
```

```
  static X x2; // Default constructor required
}

main() {
  f();
}
```

在函数f()内部定义一个静态的X类型的对象，它可以用带参数的构造函数来初始化，也可以用缺省构造函数。程序控制第一次转到对象的定义点时，而且只有第一次时，才需要执行构造函数。

2. 静态对象的析构函数

静态对象的析构函数（包括静态存储的所有对象，不仅仅是上例中的局部静态变量）在程序从main() 块中退出时，或者标准的C库函数exit()被调用时才被调用。多数情况下main()函数的结尾也是调用exit()来结束程序的。这意味着在析构函数内部使用exit()是很危险的，因为这可能陷入一个死循环中。但如果用标准的C库函数abort()来退出程序，静态对象的析构函数并不会被调用。

我们可以用标准C库函数atexit()来指定当程序跳出main()（或调用exit()）时应执行的操作。在这种情况下，在跳出main()或调用exit()之前，用atexit()注册的函数可以在所有对象的析构函数之前被调用。

静态对象的销毁是按它们初始化时相反的顺序进行的。当然只有那些已经被创建的对象才会被销毁。幸运的是，编程系统会记录对象初始化的顺序和那些已被创建的对象。全局对象总是在main()执行之前被创建了，所以最后一条语句只对函数局部的静态对象起作用。如果一个包含静态对象的函数从未被调用过，那么这个对象的构造函数也就不会执行，这样自然也不会执行析构函数。请看下例：

```
//: STATDEST.CPP -- Static object destructors
#include <fstream.h>
ofstream out("statdest.out"); // Trace file

class obj {
  char c; // Identifier
public:
  obj(char C) : c(C) {
    out << "obj::obj() for " << c << endl;
  }
  ~obj() {
    out << "obj::~obj() for " << c << endl;
  }
};

obj A('A'); // Global (static storage)
// Constructor & destructor always called

void f() {
  static obj B('B');
}

void g() {
  static obj C('C');
```

```
}

main() {
  out << "inside main()" << endl;
  f(); // Calls static constructor for B
  // g() not called
  out << "leaving main()" << endl;
}
```

在obj中，字符c的作用就象一个标识符，构造函数和析构函数就可以显示出当前正在操作的对象信息。而A是一个全局的obj类的对象，所以构造函数总是在main()函数之前就被调用。但函数f()的内部的静态obj类对象B和函数g()内部的静态对象C的构造函数只在这些函数被调用时才起作用。

为了说明哪些构造函数与析构函数被调用，在main()中只调用了f()，程序的输出结果为：

```
obj::obj() for A
inside main()
obj::obj() for B
leaving main()
obj::~obj() for B
obj::~obj() for A
```

对象A的构造函数在进入main()函数之前即被调用，而B的构造函数只是因为f()的调用而调用。当从main()函数中退出时，所有被创建的对象的析构函数按创建时相反的顺序被调用。这意味着如果g()被调用，对象B和C的析构函数的调用顺序依赖于g()和f()的调用顺序。

注意跟踪文件ofstream的对象out也是一个静态对象，它的定义（与 extern声明意义相反）应该出现在文件的一开始，在使用 out之前，这一点很重要，否则我们就可能在一个对象初始化之前使用它。

在C++中，全局静态对象的构造函数是在 main()之前调用的，所以我们现在有了一个在进入main()之前执行一段代码的简单的、可移植的方法，并且可以在退出 main()之后用析构函数执行代码。在C中要做到这一点，我们不得不熟悉编译器开发商的汇编语言的开始代码。

9.1.2 控制连接

一般情况下，在文件范围内的所有名字（既不嵌套在类或函数中的名字）对程序中的所有编译单元来说都是可见的。这就是所谓的外部连接，因为在连接时这个名字对连接器来说是可见的，外部的编译单元、全局变量和普通函数都有外部连接。

有时我们可能想限制一个名字的可见性。想让一个变量在文件范围内是可见的，这样这个文件中的所有函数都可以使用它，但不想让这个文件之外的函数看到或访问该变量，或不想这个变量的名字与外部的标识符相冲突。

在文件范围内，一个被明确声明为static的对象或函数的名字对编译单元（用本书的术语来说也就是出现声明的.CPP文件）来说是局部变量；这些名字有内部连接。这意味着我们可以在其他的编译单元中使用同样的名字，而不会发生名字冲突。

内部连接的一个好处是这个名字可以放在一个头文件中而不用担心连接时发生冲突。那些通常放在头文件里的名字，像常量、内联函数（ inline function），在缺省情况下都是内部连接的（当然常量只有在C++中缺省情况下是内部连接的，在C中它缺省为外部连接）。注意连接只引用那些在连接/装载期间有地址的成员，因此类声明和局部变量并没有连接。

• 冲突问题

下面例子说明了static的两个含义怎样彼此交叉的。所有的全局对象都是隐含为静态存储类，所以如果我们定义（在文件范围）

```
int a=0;
```

则a被存储在程序的静态数据区，在进入main()函数之前，a即已初始化了。另外，a对全局都是可见的，包括所有的编译单元。用可见性术语，static（只在编译单元内可见）的反义是extern，它表示这个名字对所有的编译单元都是可见的。所以上面的定义和下面的定义是相同的。

```
extern int a=0;
```

但如果这样定义：

```
static int a=0;
```

我们只不过改变了a的可见性，现在a成了一个内部连接。但存储类型没有改变——对象总是驻留在静态数据区，而不管是static还是extern。

一旦进入局部变量，static就不会再改变变量的可见性（这时 extern是没有意义的），而只是改变变量的存储类型。

对函数名，static和extern只会改变它的可见性，所以如果说：

```
extern void f()；
```

它和没有修饰时的声明是一样的：

```
void f()；
```

如果定义：

```
static void f()；
```

它意味着f()只在本编译单元内是可见的，这有时称作文件静态。

9.1.3 其他的存储类型指定符

我们会看到static和extern用得很普遍。另外还有两个存储类型指定符，这两种用得较少。一个是auto,人们几乎不用它，因为它告诉编译器这是一个局部变量，实际上编译器总是可以从变量定义时的上下文中判断出这是一个局部变量。所以 auto是多余的。还有一个是register，它也是局部变量，但它告诉编译器这个特殊的变量要经常用到，所以编译器应该尽可能地让它保存在寄存器中。它用于优化代码。各种编译器对这种类型的变量处理方式也不尽相同，它们有时会忽略这种存储类型的指定。一般，如果要用到这个变量的地址，register指定符通常都会被忽略。应该避免用register类型，因为编译器在优化代码方面通常比我们做得更好。

9.2 名字空间

虽然名字可以在类中被嵌套，但全局函数、全局变量以及类的名字还是在同一个名字空间中。虽然static关键字可以使变量和函数内部连接（使它们的文件静态），但在一个大项目中，如果对全局的名字空间缺乏控制就会引起很多问题。为了解决这些问题，开发商常常使用冗长、难懂的名字，以使冲突减少，但这样我们不得不一个一个地敲这些名字（typedef常常用来简化这些名字）。这不是一个很好的解决方法。

我们可以用C++的名字空间特征（我们的编译器可能还没有实现这一特征，请查阅技术文档），把一个全局名字空间分成多个可管理的小空间。名字空间的关键字，像 class,struct,enum和union一样，把它们的成员的名字放到了不同的空间中去 ,尽管其他的关键字有其他的目的，

但namespace唯一的目的是产生一个新的名字空间。

9.2.1 产生一个名字空间

名字空间的产生与一个类的产生非常相似：

```
namespace MyLib {
  // Declarations
}
```

这就产生了一个新的名字空间，其中包含了各种声明。 namespace与class、struct、union和enum有着明显的区别：

1) namespace只能在全局范畴定义，但它们之间可以互相嵌套。

2) 在namespace定义的结尾，右大括号的后面不必要跟一个分号。

3) 一个namespace可以在多个头文件中用一个标识符来定义，就好象重复定义一个类一样。

```
//: HEADER1.H
namespace MyLib {
  extern int X;
  void f();
  // ...
}
//: HEADER2.H
// Add more names to MyLib
namespace MyLib { // NOT a redefinition!
  extern int Y;
  void g();
  // ...
}
```

4) 一个namespace的名字可以用另一个名字来作它的别名，这样我们就不必敲打那些开发商提供的冗长的名字了。

```
namespace BobsSuperDuperLibrary {
  class widget { /* ... */ };
  class poppit { /* ... */ };
  // ...
}
// Too much to type! I'll alias it:
namespace Bob = BobsSuperDuperLibrary;
```

5) 我们不能像类那样去创建一个名字空间的实例。

1. 未命名的名字空间

每个编译单元都可包含一个未命名的名字空间 —— 在namespace关键字后面没有标识符。

```
namespace {
  class arm  { /* ... */ };
  class leg  { /* ... */ };
  class head { /* ... */ };
  class robot {
```

```
    arm Arm[4];
    leg Leg[16];
    head Head[3];
    // ...
  } Xanthan;
  int i, j, k;
}
```

在编译单元内，这个空间中的名字自动而无限制地有效。每个编译单元要确保只有一个未命名的名字空间。如果把一个局部名字放在一个未命名的名字空间中，无需加上 static说明就可以让它们作内部连接。

2. 友元

可以在一个名字空间的类定义之内插入一个 friend 声明：

```
namespace me {
  class us {
    //...
    friend you();
  };
}
```

这样函数you()就成了名字空间me的一个成员。

9.2.2 使用名字空间

可以用两种方法在一个名字空间引用同一个名字：一种是用范围分解运算符，还有一种是用using关键字。

1. 范围分解

名字空间中的任何命名都可以用范围分解运算符明确指定，就像引用一个类中的名字一样：

```
namespace X {
  class Y {
    static int i;
  public:
    void f();
  };
  class Z;
  void foo();
}

int X::Y::i = 9;

class X::Z {
  int u, v, w;
public:
  Z(int I);
  int g();
};
```

```
X::Z::Z(int I) { u = v = w = I; }
int X::Z::g() { return u = v = w = 0; }

void X::foo() {
  X::Z a(1);
  a.g();
}
```

到目前为止，名字空间看上去很像一个类。

2. using指令

用using 关键字可以让我们立即输入整个名字空间，摆脱输入一个名字空间中标识符的烦恼。这种using和namespace关键字的搭配使用叫作using 指令。using 关键字在当前范围内直接声明了名字空间中的所有的名字，所以可以很方便地使用这些无限制的名字：

```
namespace math {
  enum sign { positive, negative };
  class integer {
    int i;
    sign s;
  public:
    integer(int I = 0)
      : i(I),
        s(i >= 0 ? positive : negative)
    {}
    sign Sign() { return s; }
    void Sign(sign S) { s = S; }
    // ...
  };
  integer A, B, C;
  integer divide(integer, integer);
  // ...
}
```

现在可以在函数内部声明math中的所有名字，但允许这些名字嵌套在函数中。

```
void arithmetic() {
  using namespace math;
  integer X;
  X.Sign(positive);
}
```

如果不用using指令，这个名字空间的所有名字都需要完全限定。

using 指令有一个缺点，那就是看起来不那么直观， using指令引入名字可见性的范围是在创建using的地方。但我们可以使来自using 指令的名字暂时无效，就像它们已经被声明为这个范围的全局名一样。

```
void q() {
  using namespace math;
  integer A; // Hides math::A;
```

```
    A.Sign(negative);
    math::A.Sign(positive);
  }
```

如果有第二个名字空间

```
  namespace calculation {
    class integer {};
    integer divide(integer, integer);
    // ...
  }
```

这个名字空间也用using指令来引入，就可能产生冲突。这种二义性出现在名字的使用时，而不是在using指令使用时。

```
  void s() {
    using namespace math;
    using namespace calculation;
    // Everything's ok until:
    divide(1, 2); // Ambiguity
  }
```

这样，即使永远不产生二义性，写using指令引入带名字冲突的名字空间也是可能的。

3. using声明

可以用using声明一次性引入名字到当前范围内。这种方法不像 using指令那样把那些名字当成当前范围的全局名来看待，而是在当前范围之内进行一个声明，这就意味着在这个范围内它可以废弃来自using指令的名字。

```
  namespace U {
    void f();
    void g();
  }

  namespace V {
    void f();
    void g();
  }

  void func() {
    using namespace U; // Using directive
    using V::f; // Using declaration
    f(); // Calls V::f();
    U::f(); // Must fully qualify to call
  }
```

using声明给出了标识符的完整的名字，但没有了类型方面的信息。也就是说，如果名字空间中包含了一组用相同名字重载的函数， using声明就声明了这个重载的集合内的所有函数。

可以把using声明放在任何一般的声明可以出现的地方。 using声明与普通声明只有一点不同：using声明可以引起一个函数用相同的参数类型来重载（这在一般的重载中是不允许的）。

当然这种不确定性要到使用时才表现出来，而不是在声明时。

using声明也可以出现在一个名字空间内，其作用与在其他地方时一样：

```
namespace Q {
  using U::f;
  using V::g;
  // ...
}

void m() {
  using namespace Q;
  f(); // calls U::f();
  g(); // calls V::g();
}
```

一个using 声明是一个别名，它允许我们在不同的名字空间声明同样的函数。如果我们不想由于引入不同名字空间的函数而导致重复定义一个函数时，可以用 using声明，它不会引起任何不确定性和重复。

9.3 C++中的静态成员

有时需要为某个类的所有对象分配一个单一的存储空间。在 C语言中，可以用全局变量，但这样很不安全。全局数据可以被任何人修改，而且，在一个项目中，它很容易与其他的名字相冲突。如果可以把一个数据当成全局变量那样去存储，但又被隐藏在类的内部，并且清楚地与这个类相联系，这种处理方法当然是最理想的了。

这一点可以用类的静态数据成员来实现。类的静态成员拥有一块单独的存储区，而不管我们创建了多少个该类的对象。所有这些对象的静态数据成员都共享这一块静态存储空间，这就为这些对象提供了一种互相通信的方法。但静态数据属于类，它的名字只在类的范围内有效，并且可以是public（公有的）、private（私有的）或者protected（保护的）。

9.3.1 定义静态数据成员的存储

因为类的静态数据成员有着单一的存储空间而不管产生了多少个对象，所以存储空间必须定义在一个单一的地方。当然以前有些编译器会分配存储空间，但现在编译器不会分配存储空间。如果一个静态数据成员被声明但没有定义时，连接器会报告一个错误。

定义必须出现在类的外部（不允许内联）而且只能定义一次，因此它通常放在一个类的实现文件中。这种规定常常让人感到很麻烦，但实际上它是很合理的。比方说：

```
class A {
  static int i;
public:
  //...
};
```

之后，在定义文件中，

```
int A::i=1;
```

如果定义了一个普通的全局变量，可以写：

```
int i=1;
```

在这里，类名和范围分解运算符用于指定了i的范围。

有些人对A::i是私有的这点感到疑惑不解，还有一些事似乎被公开地处理。这不是破坏了类结构的保护性吗？有两个原因可以保证它绝对的安全。第一，这些变量的初始化唯一合法是在定义时。事实上，如果静态数据成员是一个带构造函数的对象时，可以调用构造函数来代替“=”操作符。第二，一旦这些数据被定义了，终端用户就不能再定义它——否则连接器会报告错误。而且这个类的创建被迫产生这个定义，否则这些代码在测试时无法连接。这就保证了定义只出现一次并且它是由类的构造者来控制的。

一个静态成员的初始化表达式是在一个类的范围内，请看下例：

```
//: STATINIT.CPP -- Scope of static initializer
#include <iostream.h>

int x = 100;

class withStatic {
  static int x;
  static int y;
public:
  void print() const {
    cout << "withStatic::x = " << x << endl;
    cout << "withStatic::y = " << y << endl;
  }
};

int withStatic::x = 1;
int withStatic::y = x + 1;
// WithStatic::x NOT ::x

main() {
  withStatic WS;
  WS.print();
}
```

这里，withStatic::限定符把withStatic的范围扩展到全部定义。

1. 静态数组的初始化

我们不仅可以产生静态常量对象，而且可以产生静态数组对象，包括常量数组与非常量数组，下面是初始化一个静态数组的例子：

```
//: STATARRY.CPP -- Initializing static arrays

class Values {
  static const int size;
  static const float table[4];
  static char letters[10];
};
```

```
const int Values::size = 100;

const float Values::table[4] = {
  1.1, 2.2, 3.3, 4.4
};

char Values::letters[10] = {
  'a', 'b', 'c', 'd', 'e',
  'f', 'g', 'h', 'i', 'j'
};

main() {}
```

对所有的静态数据成员，我们必须提供一个单一的外部定义。这些定义必须有内部连接，所以可以放在头文件中。初始化静态数组的方法与其他集合类型的初始化一样，但不能用自动计数。除此之外，在类定义结束时，编译器必须知道足够的类信息来创建对象，包括所有成员的精确大小。

2. 类中的编译时常量

在第7章中，我们介绍了用枚举型来创建一个编译时常量（一种被计算出来的常量表达式，如数组大小）的方法，它作为一个类的局部常量。尽管这种方法的使用很普遍，但常被称作“ enum hack ”，因为它使枚举丧失了本来的作用。

为了用一种更好的方法来完成同样的事，我们可以在一个类中使用一个静态常量（我们的编译器可能还不支持这一技巧，请查阅随机文档）。因为它既是常量（它不会改变）又是静态的（整个类中只有唯一的一个定义点），所以一个类中的静态常量可被用作一个编译时常量，如下例：

```
class X {
  static const int size;
  int array[size];
public:
  // ...
};

const int X::size = 100; // definition
```

如果我们在类中的一个常量表达式用到静态常量，那么这个静态常量的定义应该出现在这个类的任何实例或类的成员函数定义之前（也许在头文件中）。和一个内部数据类型的全局常量一样，它并不会为常量分配存储空间，又由于它是内部连接的，所以也不会产生冲突。

这种方法的另一个好处是任何预定义类型都可以作为一个静态常量成员，而用 enum时，只能使用整型值。

9.3.2 嵌套类和局部类

可以很容易地把一个静态数据成员放在一个嵌套内中。这样的成员的定义显然是上节中情况的扩展——我们只须用另一种级别的指定。然而在局部类（在函数内部定义的类）中不能有

静态数据成员。如下例：

```
//: LOCAL.CPP -- Static members & local classes
#include <iostream.h>

// Nested class CAN have static data members:
class outer {
  class inner {
    static int i; // OK
  };
};

int outer::inner::i = 47;

// Local class cannot have static data members:
void f() {
  class foo {
  public:
//! static int i;  // Error
    // (how would you define i?)
  } x;
}

main() {}
```

我们可以看到一个局部类中有静态成员的直接问题。为了定义它，怎样才能在文件范围描述一个数据呢？实际上局部类很少使用。

9.3.3 静态成员函数

像静态数据成员一样，我们也可以创建一个静态成员函数，它为类的全体服务而不是为一个类的部分对象服务。这样就不需要定义一个全局函数，减少了全局或局部名字空间的占用，把这个函数移到了类的内部。当产生一个静态成员函数时，也就表达了与一个特定类的联系。

静态成员函数不能访问一般的数据成员，它只能访问静态数据成员，也只能调用其他的静态成员函数。通常，当前对象的地址（this）是被隐含地传递到被调用的函数的。但一个静态成员函数没有this，所以它无法访问一般的成员函数。这样使用静态成员函数在速度上可以比全局函数有少许的增长，它不仅没有传递 this所需的额外的花费，而且还有使函数在类内的好处。

用static关键字指定了一个类的所有对象占有相同的一块存储空间，函数可以并行使用它，这意味着一个局部变量只有一个拷贝，函数每次调用都使用它。

下面是一个静态数据成员和静态成员函数如何在一起使用的例子：

```
//: SFUNC.CPP -- Static member functions

class X {
  int i;
  static int j;
```

```
public:
  X(int I = 0) : i(I) {
     // Non-static member function can access
     // Static member function or data:
    j = i;
  }
  int val() const { return i; }
  static int incr() {
    //! i++; // Error: static member function
    // Cannot access non-static member data
    return ++j;
  }
  static int f() {
    //! val(); // Error: static member function
    // Cannot access non-static member function
    return incr(); // OK -- calls static
  }
};

int X::j = 0;

main() {
  X x;
  X* xp = &x;
  x.f();
  xp->f();
  X::f(); // Only works with static members
}
```

因为静态成员函数没有this指针，所以它不能访问非静态的数据成员，也不能调用非静态的成员函数，这些函数要用到this指针)。

注意在main()中一个静态成员可以用点或箭头来选取，把那个函数与一个对象联系起来，但也可以不与对象相连（因为一个静态成员是与一个类相联，而不是与一个特定的对象相连），用类的名字和范围分解运算符。

这是一个有趣的特点：因为静态成员对象的初始化方法，我们可以把上述类的一个静态数据成员放到那个类的内部。下面是一个例子，它把构造函数变成私有的，这样 egg类只有一个唯一的对象存在，我们可以访问那个对象，但不能产生任何新的egg对象。

```
//: SELFMEM.CPP -- Static member of same type
// Ensures only one object of this type exists.
// Also referred to as a "singleton" pattern.
#include <iostream.h>

class egg {
  static egg E;
  int i;
```

```
  egg(int I) : i(I) {}
public:
  static egg* instance() { return &E; }
  int val() { return i; }
};

egg egg::E(47);

main() {
//!  egg x(1); // error -- can't create an egg
  // You can access the single instance:
  cout << egg::instance()->val() << endl;
}
```

E的初始化出现在类的声明完成后，所以编译器已有足够的信息为对象分配空间并调用构造函数。

9.4 静态初始化的依赖因素

在一个指定的编译单元中，静态对象的初始化顺序严格按照对象在该单元中定义出现的顺序。而清除的顺序则与初始化的顺序正好相反。

当然在多个编译单元之间没有严格的初始化顺序，也没有办法来指定这种顺序。这可能会引起不小问题。下面的例子如果一个文件包含上述情况就会立即引起灾难（它会暂停操作系统的运行，中止复杂的进程）。

```
// first file
#include <fstream.h>
ofstream out("out.txt");
```

另一个文件在它的初始表达式之一中用到了 out 对象：

```
// second file
#include <fstream.h>
extern ofstream out;
class oof {
public:
  oof() { out << "barf"; }
} OOF;
```

这个程序可能运行，也可能不运行。如果在建立可执行文件时第一个文件先初始化，那么就不会有问题，但如果第二个文件先初始化，oof的构造函数依赖out的存在，而此时out还没有创建，于是引起混乱。这只是一个互相依赖的静态对象初始化的问题，因为当我们进入 main()时，所有静态对象的构造函数都已经被调用了。

在ARM [1]中可以看到一个更丧气的例子，在一个文件中：

```
extern int y;
int x = y + 1;
```

[1] The Annotated C++ Reference Manual,Bjarne Stroustrup和Margaret Ellis著，1990年，pp.20-21。

在另一个文件中

```
extern int x;
int y = x + 1;
```

对所有的静态对象，连接装载系统在程序员指定的动态初始化发生前保证一个静态成员初始化为零。在前一个例子中，fstream out 对象的存储空间赋零并没有特殊的意思，所以它在构造函数调用前确实是未定义的。然而，对内部数据类型，初始化为零是有意义的，所以如果文件按上面的顺序被初始化，y开始被初始化为零，所以x变成1，而后y被动态初始化为2。然而，如果初始化的顺序颠倒过来，x被静态初始化为零,y被初始化为1，而后x被初始化为2。

程序员必须意识到这些，因为他们可能会在编程时遇到互相依赖的静态变量的初始化问题，程序可能在一个平台上工作正常，把它移到另一个编译环境时，突然莫名其妙地不工作了。

怎么办

有三种方法来处理这一问题：

1) 不用它，避免初始化时的互相依赖。这是最好的解决方法。

2) 如果实在要用，就把那些关键的静态对象的定义放在一个文件中，这样我们只要让它们在文件中顺序正确就可以保证它们正确的初始化。

3) 如果我们确信把静态对象放在几个编译单元中是不可避免的（比方在编写一个库时，我们无法控制那些使用该库的程序员）这时我们可用由 Jerry Schwarz在创建iostream库（因为cin,cout和cerr的定义是在不同的文件中）时提供的一种技术。

这一技术要求在库头文件中加上一个额外的类。这个类负责库中静态对象的动态初始化。下面是一个简单的例子：

```
//: DEPEND.H -- Static initialization technique
#ifndef DEPEND_H_
#define DEPEND_H_
#include <iostream.h>
extern int x; // Delarations, not definitions
extern int y;

class initializer {
  static int init_count;
public:
  initializer() {
    cout << "initializer()" << endl;
    // Initialize first time only
    if(init_count++ == 0) {
      cout << "performing initialization"
           << endl;
      x = 100;
      y = 200;
    }
  }
~initializer() {
  cout << "~initializer()" << endl;
```

```
    // Clean up last time only
    if(--init_count == 0) {
      cout << "performing cleanup" << endl;
      // Any necessary cleanup here
    }
  }
};

// The following creates one object in each
// file where DEPEND.H is included, but that
// object is only visible within that file:
static initializer init;

#endif // DEPEND_H_
```

x、y的声明只是表明这些对象的存在，并没有为它们分配存储空间。然而 initializer init 的定义为每个包含此头文件的文件分配那些对象的空间，因为名字是 static的（这里控制可见性而不是指定存储类型，因为缺省时是在文件范围内）它只在本编译单元可见，所以连接器不会报告一个多重定义错误。

下面是一个包含x、y和init_count定义的文件：

```
//: DEPDEFS.CPP -- Definitions for DEPEND.H
#include "depend.h"
// Static initialization will force
// all these values to zero:
int x;
int y;
int initializer::init_count;
```

（当然，一个文件的init静态实例也放在这个文件中）假设库的使用者产生了两个其他的文件：

```
//: DEPEND.CPP -- Static initialization
#include "depend.h"
```

和

```
//: DEPEND2.CPP -- Static initialization
#include "depend.h"

main() {
  cout << "inside main()" << endl;
  cout << "leaving main()" << endl;
}
```

现在哪个编译单元先初始化都没有关系。当第一次包含 DEPEND.H的编译单元被初始化时，init_count为零，这时初始化就已经完成了（这是由于内部类型的全局变量在动态初始化之前都被设置为零）。对其余的编译单元，初始化会跳过去。清除按相反的顺序，且 ~initializer()可确

保它只发生一次。

这个例子用内部类型作为全局静态对象，这种方法也可以用于类，但其对象必须用initializer动态初始化。一种方法就是创建一个没有构造函数和析构函数的类，但用不同的名字的成员函数来初始化和清除这个类。当然更常用的做法是在 initializer()函数中，设定指向对象的指针，并在堆中动态创建它们。这要用到两个C++的关键字new和delete，第12章中介绍。

9.5 转换连接指定

如果C++中编写一个程序需要用到C库，那该怎么办呢？如果这样声明一个C函数：

```
float f(int a,char b);
```

C++的编译器就会将这个名字变成像 _f_int_int之类的东西以支持函数重载（和类型安全连接）。然而，C编译器编译的库一般不做这样的转换，所以它的内部名为 _f。这样，连接器将无法解决我们C++对f（）的调用。

C++中提供了一个连接转换指定，它是通过重载 extern关键字来实现的。extern后跟一个字符串来指定我们想声明的函数的连接类型，后面是函数声明。

```
extern "C" float f(int a,char b);
```

这就告诉编译器 f()是C连接，这样就不会转换函数名。标准的连接类型指定符有“ C ”和“ C++ ”两种，但编译器开发商可选择用同样的方法支持其他语言。

如果我们有一组转换连接的声明，可以把它们放在花括号内：

```
extern "C" {
  float f(int a, char b);
  double d(int a, char b);
}
```

或在头文件中：

```
extern "C" {
#include "myheader.h"
}
```

多数C++编译器开发商在他们的头文件中处理转换连接指定，包括 C和C++，所以我们不用担心它们。

虽然标准的C++只支持“ C ”和“ C++ ”两种连接转换指定，但用同样的方法可以实现对其他语言的支持。

9.6 小结

static关键字很容易使人糊涂，因为它有时控制存储分配，而有时控制一个名字的可见性和连接。

随着C++名字空间的引入，我们有了更好的、更灵活的方法来控制一个大项目中名字的增长。

在类的内部使用static是在全程序中控制名字的另一种方法。这些名字不会与全局名冲突，并且可见性和访问也限制在程序内部，使我们在维护我们的代码时能有更多的控制。

9.7 练习

1. 创建一个带整型数组的类。在类内部用未标识的枚举变量来设置数组的长度。增加一个

const int 变量，并在构造函数初始化表达式表中初始化。增加一个 static int 成员变量并用特定值来初始化。增加一个内联（inline）构造函数和一个内联（inline）型的print()函数来显示数组中的全部值，并在这两函数内调用静态成员函数。

2. STATDEST.CPP中，在main()内用不同的顺序调用f()、g()来检验构造函数与析构函数的调用顺序，我们的编译器能正确地编译它们吗？

3. 在STATDEST.CPP中，把out的定义变为一个extern声明，并把实际定义放到A（它的构造函数obj传送信息给out）的定义之后，测试我们的机器是怎样进行缺省错误处理的。当我们运行程序时确保没有其他重要程序在运行，否则我们的机器会出现错误。

4. 创建一个类，它的析构函数显示一条信息，然后调用 exit()。创建这个类的一个全局静态对象，看看会发生什么？

5. 修改第7章的VOLATILE.CPP，使comm::isr()作为一个中断服务例程来运行。

第10章　引用和拷贝构造函数

引用是C++的一个特征，它就像能自动被编译器逆向引用的常量型指针一样。

虽然Pascal语言中也有引用，但C++中引用的思想来自于Algol语言。引用是支持C++运算符重载语法的基础（见 第11章），也为函数参数传入和传出的控制提供了便利。

本章首先看一下C和C++的指针的差异，然后介绍引用。但本章的大部分内容将研究令人迷糊的C++新的编程问题：拷贝构造函数 (copy-constructor)。它是特殊的构造函数，需要用引用(&)来实现从现有的相同类型的对象产生新的对象。编译器用拷贝构造函数通过传值的方式来传递和返回对象。

本章最后将阐述有点难以理解的C++的指向成员的指针(pointer-to-member)的概念。

10.1　C++中的指针

C和C++指针的最重要的区别，在于C++是一种类型要求更强的语言。就void*而言，这一点表现得更加突出。C不允许随便地把一个类型的指针指派给另一个类型，但允许通过 void*来实现。例如：

```
bird*  b;
rock*  r;
void*  v;
v = r;
b = v;
```

C++不允许这样做，其编译器将会给出一个出错信息。如果真的想这样做，必须显式地使用映射，通知编译器和读者（见18章C++改进的映射语法）。

10.2　C++中的引用

引用(&)像一个自动能被编译器逆向引用的常量型指针。它通常用于函数的参数表中和函数的返回值，但也可以独立使用。例如：

```
int  x;
int  & r = x;
```

当创建了一个引用时，引用必须被初始化指向一个存在的对象。但也可以这样写：

```
int  & q = 12;
```

这里，编译器分派了一个存储单元，它的值被初始化为 12，这样这个引用就和这个存储单元联系上了。要点是任何引用必须和存储单元联系。但访问引用时，就是在访问那个存储单元。因而，如果这样写：

```
int x=0;
int & a = x;
a++;
```

增加a事实上是增加x。考虑一个引用的最简单的方法是把它当作一个奇特的指针。这个指针的一个优点是不必怀疑它是否被初始化了（编译器强迫它初始化），也不必知道怎样对它逆

向引用（这由编译器做）。

使用引用时有一定的规则：

1) 当引用被创建时，它必须被初始化。（指针则可以在任何时候被初始化。）

2) 一旦一个引用被初始化为指向一个对象，它就不能被改变为对另一个对象的引用。（指针则可以在任何时候指向另一个对象。）

3) 不可能有NULL引用。必须确保引用是和一块合法的存储单元关连。

10.2.1 函数中的引用

最经常看见引用的地方是在函数参数和返回值中。当引用被用作函数参数时，函数内任何对引用的更改将对函数外的参数改变。当然，可以通过传递一个指针来做相同的事情，但引用具有更清晰的语法。（如果愿意的话，可以把引用看作一个使语法更加便利的工具。）

如果从函数中返回一个引用，必须像从函数中返回一个指针一样对待。当函数返回时，无论引用关连的是什么都不应该离开，否则，将不知道指向哪一个内存区域。

这儿有一个例子：

```
//: REFRNCE.CPP -- Simple C++ references

int* f(int* x) {
  (*x)++;
  return x; // Safe; x is outside this scope
}

int& g(int& x) {
  x++; // Same effect as in f()
  return x; // Safe; outside this scope
}

int& h() {
  int q;
//!  return q;  // Error
  static int x;
  return x; // Safe; x lives outside scope
}

main() {
  int A = 0;
  f(&A); // Ugly (but explicit)
  g(A);  // Clean (but hidden)
}
```

对函数f( )的调用缺乏使用引用的方便性和清晰性，但很清楚这是传递一个地址。在函数 g ( )的调用中，地址通过引用被传递，但表面上看不出来。

1. 常量引用

仅当在REFRNCE.CPP例程中的参数是非常量对象时，这个引用参数才能工作。如果是常量对象，函数g ( )将不接受这个参数，这样做是一件好事，因为这个函数将改变外部参数。如

果我们知道这函数不妨碍对象的不变性的话，让这个参数是一个常量引用将允许这个函数在任何情况下使用。这意味着，对于内部类型，这个函数不会改变参数，而对于用户定义的类型，该函数只能调用常量成员函数，而且不应当改变任何公共的数据成员。

在函数参数中使用常量引用特别重要。这是因为我们的函数也许会接受临时的对象，这个临时对象是由另一个函数的返回值创立或由函数使用者显式地创立的。临时对象总是不变的，因此如果不使用常量引用，参数将不会被编译器接受。看下面一个非常简单的例子：

```
//: PASCONST.CPP -- Passing references as const

void f(int&) {}
void g(const int&) {}

main() {
//!  f(1); // Error
  g(1);
}
```

调用f(1)会产生一个编译错误，这是因为编译器必须首先建立一个引用。即编译器为一个int类型分派存储单元，同时将其初始化为1并为其产生一个地址和引用捆绑在一起。存储的内容必须是常量，因为改变它将使之变得没有意义。对于所有的临时对象，必须同样假设它们是不可存取的。当改变这种数据的时候，编译器会指出错误，这是非常有用的提示，因为这个改变会导致信息丢失。

2. 指针引用

在C语言中，如果想改变指针本身而不是它所指向的内容，函数声明可能像这样：

```
void f (int**);
```

传递它时，必须取得指针的地址，像下面的例子：

```
int I = 47;
int* ip = &I;
f (&ip);
```

对于C++中的引用，语法清晰多了。函数参数变成指针的引用，用不着取得指针的地址。通过运行下面的程序，将会看到指针本身增加了，而不是它指向的内容增加了。

```
//: REFPTR.CPP -- Reference to pointer
#include <iostream.h>

void increment(int*& i) { i++; }

main() {
  int* i = 0;
  cout << "i = " << i << endl;
  increment(i);
  cout << "i = " << i << endl;
}
```

10.2.2 参数传递准则

当给函数传递参数时，人们习惯上应该是通过常量引用来传递。虽然最初看起来似乎仅出

于对效率的考虑（通常在设计和装配程序时并不考虑效率），但像本章以后部分介绍的，这里将会存在很多的危险。拷贝构造函数需要通过值来传递对象，但这并不总是可行的。

这种简单习惯可以大大提高效率：传值方式需要调用构造函数和析构函数，然而如果不想改变参数，则可通过常量引用传递，它仅需要将地址压栈。

事实上，只有一种情况不适合用传递地址方式，这就是当传值是唯一安全的途径，否则将会破坏对象时（而不是修改外部对象，这不是调用者通常期望的）。这是下一节的主题。

10.3 拷贝构造函数

介绍了C++中的引用的基本概念后，我们将讲述一个更令人混淆的概念：拷贝构造函数，它常被称为X(X&)（“X引用的X”）。在函数调用时，这个构造函数是控制通过传值方式传递和返回用户定义类型的根本所在。

10.3.1 传值方式传递和返回

为了理解拷贝构造函数的需要，我们来看一下C语言在调用函数时处理通过传值方式传递和返回变量的方法。

```
int f (int x, char c);
int g = f (a,b);
```

编译器如何知道怎样传递和返回这些变量？其实它天生就知道！因为它必须处理的类型范围是如此之小（char，int，float，double和它们的变量），这些信息都被内置在编译器中。

如果能了解编译器怎样产生汇编代码和怎样确定调用函数 f ( )产生语句的，我们就能得到下面的等价物：

```
push b
push a
call f ( )
add sp,4
mov g, register a
```

这个代码已被认真整理过，使之具有普遍意义——b和a的表达式根据变量是全局变量（在这种情况下它们是_b和_a）或局部变量（编译器将在堆栈指针上对其索引）将有差异。 g表达式也是这样。对f( )调用的形式取决于我们的name-mangling方案，“寄存器 a”取决于CPU寄存器在我们的汇编程序中是如何命名的。但不管代码如何，逻辑是相同的。

在C和C++中，参数是从右向左进栈，然后调用函数，调用代码负责清理栈中的参数（这一点说明了add sp,4的作用）。但是要注意，通过传值方式传递参数时，编译器简单地将参数拷贝压栈——编译器知道拷贝有多大，并知道为压栈的参数产生正确的拷贝。

f( )的返回值放在寄存器中。编译器同样知道返回值的类型，因为这个类型是内置于语言中的，于是编译器可以通过把返回值放在寄存器中返回它。拷贝这个值的比特位等同于拷贝对象。

1. 传递和返回大对象

现在来考虑用户定义的类型。如果创建了一个类，我们希望传递该类的一个对象，编译器怎么知道做什么？这是编译器的作者所不知的非内部数据类型，是别人创建的类型。

为了研究这个问题，我们首先从一个简单的结构开始，这个结构太大以至于不能在寄存器中返回。

```
//: PASSTRUC.CPP -- Passing a big structure

struct big {
  char buf[100];
  int i;
  long d;
} B, B2;

big bigfun(big b) {
  b.i = 100; // Do something to the argument
  return b;
}

main() {
  B2 = bigfun(B);
}
```

在这里列出汇编代码输出有点复杂，因为大多数编译器使用“ helper ”函数而不是设置所有功能性内置。在main()函数中，正如我们猜测的，首先调用函数 bigfun()，整个B的内容被压栈。（我们可能发现有些编译器把B的地址和大小装入寄存器，然后调用helper函数把它压栈。）

在先前的例子中，调用函数之前要把参数压栈。然而，在 PASSTRUC.CPP中，将看到附加的动作：在函数调用之前， B2的地址压栈，虽然它明显不是一个参数。为了理解这里发生的事，必须了解当编译器调用函数时对编译器的约束。

2. 函数调用栈的框架

当编译器为函数调用产生代码时，它首先把所有的参数压栈，然后调用函数。在函数内部，产生代码，向下移动栈指针为函数局部变量提供存储单元。（在这里“ 下 ”是相对的，在压栈时，机器栈指针可能增加也可能减小。）在汇编语言CALL中，CPU把程序代码中的函数调用指令的地址压栈，所以汇编语言 RETURN可以使用这个地址返回到调用点。当然，这个地址是非常神圣的，因为没有它程序将迷失方向。这儿提供一个在 CALL后栈框架的样子，此时在函数中已为局部变量分配了存储单元。

函数参数
返回地址
局部变量

函数的其他部分产生的代码完全按照这个方法安排内存，因此它可以谨慎地从函数参数和局部变量中存取而不触及返回地址。我称函数调用过程中被函数使用的这块内存为函数框架(function frame)。另外，试图从栈中得到返回值是合乎道理的。因为编译器简单地把返回值压栈，函数可以返回一个偏移值，它告诉返回值的开始在栈中所处的位置。

3. 重入

因为在C和C++的函数支持中断，所以这将出现语言重入的难题。同时，它们也支持函数递归调用。这就意味着在程序执行的任何时候，中断都可以发生而不打乱程序。当然，编写中断服务程序（ISR）的作者负责存储和还原他所使用的所有的寄存器，但如果 ISR需要使用深

入堆栈的内存时，就要小心了。（可以把ISR看成没有参数和返回值是void的普通函数，它存储和还原CPU的状态。有些硬件事件触发一个ISR函数的调用，而不是在程序中显式地调用。）

现在来想象一下，如果调用函数试图从普通函数中返回堆栈中的值将会发生什么。因为不能触及堆栈返回地址以上任何部分，所以函数必须在返回地址下将值压栈。但当汇编语言RETURN执行时，堆栈指针必须指向返回地址（或正好位于它下面，这取决于机器。），所以恰好在RETURN语句之前，函数必须将堆栈指针向上移动，以便清除所有局部变量。如果我们试图从堆栈中的返回地址下返回数值，因为中断可能此时发生，此时是我们最易被攻击的时候。这个时候ISR将向下移动堆栈指针，保存返回地址和局部变量，这样就会覆盖掉我们的返回值。

为了解决这个问题，在调用函数之前，调用者应负责在堆栈中为返回值分配额外的存储单元。然而，C不是按照这种方法设计的，C++也一样。正如我们不久将看到的，C++编译器使用更有效的方案。

我们的下一个想法可能是在全局数据区域返回数值，但这不可行。重入意味着任何函数可以中断任何其他的函数，包括与之相同的函数。因此，如果把返回值放在全局区域，我们可能又返回到相同的函数中，这将重写返回值。对于递归也是同样道理。

唯一安全的返回场所是寄存器，问题是当寄存器没有足够大用于存放返回值时该怎么做。答案是把返回值的地址像一个函数参数一样压栈，让函数直接把返回值信息拷贝到目的地。这样做不仅解决了问题，而且效率更高。这也是在PASSTRUC.CCP中main( )中bigfun( ) 调用之前将B2的地址压栈的原因。如果看了bigfun( )的汇编输出，可以看到它存在这个隐藏的参数并在函数内完成向目的地的拷贝。

4. 位拷贝（bitcopy）与初始化

迄今为止，一切都很顺利。对于传递和返回大的简单结构有了可使用的方法。但注意我们所用的方法是从一个地方向另一个地方拷贝比特位，这对于 C着眼于变量的原始方法当然进行得很好。但在C++中，对象比一组比特位要丰富得多，因为对象具有含义。这个含义也许不能由它具有的位拷贝来很好地反映。

下面来考虑一个简单的例子：一个类在任何时候知道它存在多少个对象。从第 9章，我们了解到可以通过包含一个静态(static)数据成员的方法来做到这点。

```
//: HOWMANY.CPP -- Class counts its objects
#include <fstream.h>
ofstream out("howmany.out");

class howmany {
  static int object_count;
public:
  howmany() {
    object_count++;
  }
  static void print(const char* msg = 0) {
    if(msg) out << msg << ": ";
    out << "object_count = "
        << object_count << endl;
  }
  ~howmany() {
    object_count--;
```

```
    print("~howmany()");
  }
};

int howmany::object_count = 0;

// Pass and return BY VALUE:
howmany f(howmany x) {
  x.print("x argument inside f()");
  return x;
}

main() {
  howmany h;
  howmany::print("after construction of h");
  howmany h2 = f(h);
  howmany::print("after call to f()");
}
```

howmany类包括一个静态int类型变量和一个用以报告这个变量的静态成员函数print( )，这个函数有一个可选择的消息参数。每当一个对象产生时，构造函数增加记数，而对象销毁时，析构函数减小记数。

然而，输出并不是我们所期望的那样：

```
after construction of h: object_count = 1
x argument inside f(): object_count = 1
~howmany(): object_count = 0
after call to f(): object_count = 0
~howmany(): object_count = -1
~howmany(): object_count = -2
```

在h生成以后，对象数是1，这是对的。我们希望在f( )调用后对象数是2，因为h2也在范围内。然而，对象数是0，这意味着发生了严重的错误。这从结尾两个析构函数执行后使得对象数变为负数的事实得到确认，有些事根本就不应该发生。

让我们来看一下函数f( )通过传值方式传入参数那一处。原来的对象h存在于函数框架之外，同时在函数体内又增加了一个对象，这个对象是传值方式传入的对象的拷贝。然而，参数的传递是使用C的原始的位拷贝的概念，但C++ howmany类需要真正的初始化来维护它的完整性。所以，缺省的位拷贝不能达到预期的效果。

当局部对象出了调用的函数f( )范围时，析构函数就被调用，析构函数使object_count减小。所以，在函数外面，object_count等于0。h2对象的创建也是用位拷贝产生的，所以，构造函数在这里也没有调用。当对象h和h2出了它们的作用范围时，它们的析构函数又使object_count值变为负值。

10.3.2 拷贝构造函数

上述问题的出现是因为编译器对如何从现有的对象产生新的对象作了假定。当通过传值的

方式传递一个对象时，就创立了一个新对象，函数体内的对象是由函数体外的原来存在的对象传递的。从函数返回对象也是同样的道理。在表达式中：

```
howmany h2 = f(h);
```

先前未创立的对象h2是由函数f( )的返回值创建的，所以又从一个现有的对象中创建了一个新对象。

编译器假定我们想使用位拷贝（bitcopy）来创建对象。在许多情况下，这是可行的。但在howmany类中就行不通，因为初始化不仅仅是简单的拷贝。如果类中含有指针又将出现问题：它们指向什么内容，是否拷贝它们或它们是否与一些新的内存块相连？

幸运的是，我们可以介入这个过程，并可以防止编译器进行位拷贝 (bitcopy)。每当编译器需要从现有的对象创建新对象时，我们可以通过定义我们自己的函数做这些事。因为我们是在创建新对象，所以，这个函数应该是构造函数，并且传递给这个函数的单一参数必须是我们创立的对象的源对象。但是这个对象不能传入构造函数，因为我们试图定义处理传值方式的函数按句法构造传递一个指针是没有意义的，毕竟我们正在从现有的对象创建新对象。这里，引用就起作用了，可以使用源对象的引用。这个函数被称为拷贝构造函数，它经常被提及为 X(X&)（它是被称为X的类的外在表现）。

如果设计了拷贝构造函数，当从现有的对象创建新对象时，编译器将不使用位拷贝(bitcopy)。编译器总是调用我们的拷贝构造函数。所以，如果我们没有设计拷贝函数，编译器将做一些判断，但我们完全可以接管这个过程的控制。

现在我们可以关注HOWMANY.CPP中的问题了：

```
//: HOWMANY2.CPP -- The copy-constructor
#include <fstream.h>
#include <string.h>
ofstream out("howmany2.out");

class howmany2 {
  enum { bufsize = 30 };
  char id[bufsize]; // Object identifier
  static int object_count;
public:
  howmany2(const char* ID = 0) {
    if(ID) strncpy(id, ID, bufsize);
    else *id = 0;
    ++object_count;
    print("howmany2()");
  }
  // The copy-constructor:
  howmany2(const howmany2& h) {
    strncpy(id, h.id, bufsize);
    strncat(id, " copy", bufsize - strlen(id));
    ++object_count;
    print("howmany2(howmany2&)");
  }
  // Can't be static (printing id):
```

```
  void print(const char* msg = 0) const {
    if(msg) out << msg << endl;
    out << '\t' << id << ": "
        << "object_count = "
        << object_count << endl;
  }
  ~howmany2() {
    --object_count;
    print("~howmany2()");
  }
};

int howmany2::object_count = 0;

// Pass and return BY VALUE:
howmany2 f(howmany2 x) {
  x.print("x argument inside f()");
  out << "returning from f()" << endl;
  return x;
}

main() {
  howmany2 h("h");
  out << "entering f()" << endl;
  howmany2 h2 = f(h);
  h2.print("h2 after call to f()");
  out << "call f(), no return value" << endl;
  f(h);
  out << "after call to f()" << endl;
}
```

这儿有一些新的手法，要很好地理解。首先，字符缓冲器 id起着对象识别作用，所以可以判断被打印的信息是哪一个对象的。在构造函数内，可以设置一个标识符（通常是对象的名字）。使用标准C库函数strncpy( )把标识符拷贝给id。strncpy( )只拷贝一定数目的字符，这是为防止超出缓冲器的限度。

其次是拷贝构造函数howmany2(howmany2&)。拷贝构造函数可以仅从现有的对象创立新对象，所以，现有对象的名字被拷贝给id，id后面跟着单词“ copy ”，这样我们就能了解它是从哪里拷贝来的。注意，使用标准C库函数strncat( )拷贝字符给id也得防止超过缓冲器的限度。

在拷贝构造函数内部，对象数目会像普通构造函数一样的增加。这意味着当参数传递和返回时，我们能得到准确的对象数目。

print( )函数已经被修改，用于打印消息、对象标识符和对象数目。现在 print( )函数必须存取具体对象的id数据，所以不再是static成员函数。

在main( )函数内部，可以看到又增加了一次函数 f( ) 的调用。但这次使用了普通的C语言调用方式，忽略了函数的返回值。既然现在知道了值是如何返回的（即在函数体内，代码处理返回过程并把结果放在目的地，目的地地址作为一个隐藏的参数传递。），我们可能想知道返回

值被忽略将会发生什么事情。程序的输出将对此作出解释。

在显示输出之前，这儿提供了一个小程序，这个小程序使用 iostreams为文件加入行号：

```
//: LINENUM.CPP -- Add line numbers
#include <fstream.h>
#include <strstrea.h>
#include <stdlib.h>
#include "..\allege.h"

main(int argc, char* argv[]) {
  if(argc < 2) {
    cerr << "usage: linenum file\n"
         << "adds line numbers to file"
         << endl;
    exit(1);
  }
  strstream text;
  {
    ifstream in(argv[1]);
    allegefile(in);
    text << in.rdbuf(); // Read in whole file
  } // Close file
  ofstream out(argv[1]); // Overwrite file
  const bsz = 100;
  char buf[bsz];
  int line = 0;
  while(text.getline(buf, bsz)) {
    out.setf(ios::right, ios::adjustfield);
    out.width(2);
    out << ++line << ") " << buf << endl;
  }
}
```

整个文件被读入strstream（我们可以从中写和读），ifstream在其范围内被关闭。然后为相同的文件创建一个ofstream, ofstream重写这个文件。getline( )从strstream中一次取一行，加上行号后写回文件。

行号以右对齐方式按 2个字段宽打印，所以输出仍然按原来的方式排列。可以改变程序，在程序中加入可选择的第 2个命令行参数。这个命令行参数可以让用户选择字段宽，或可以做得更聪明些，通过计算文件行数自动决定字段宽度。

当LINENUM.CPP被应用于HOWMANY2.OUT时，结果如下：

```
1) howmany2()
2)    h: object_count = 1
3) entering f()
4) howmany2(howmany2&)
5)    h copy: object_count = 2
6) x argument inside f()
```

```
 7)    h copy: object_count = 2
 8) returning from f()
 9) howmany2(howmany2&)
10)    h copy copy: object_count = 3
11) ~howmany2()
12)    h copy: object_count = 2
13) h2 after call to f()
14)    h copy copy: object_count = 2
15) call f(), no return value
16) howmany2(howmany2&)
17)    h copy: object_count = 3
18) x argument inside f()
19)    h copy: object_count = 3
20) returning from f()
21) howmany2(howmany2&)
22)    h copy copy: object_count = 4
23) ~howmany2()
24)    h copy: object_count = 3
25) ~howmany2()
26)    h copy copy: object_count = 2
27) after call to f()
28) ~howmany2()
29)    h copy copy: object_count = 1
30) ~howmany2()
31)    h: object_count = 0
```

正如我们所希望的，第一件发生的事是为h调用普通的构造函数，对象数增加为1。但在进入函数f( )时，拷贝构造函数被编译器调用完成传值过程。在f( )内创建了一个新对象，它是h的拷贝（因此被称为“ h拷贝 ”），所以对象数变成2，这是拷贝构造函数作用的结果。

第8行显示了从f( )返回的开始情况。但在局部变量“ h拷贝 ”销毁以前（在函数结尾这个局部变量便出了范围），它必须被拷入返回值，也就是h2。先前未创建的对象（h2）是从现有的对象（在函数f( )内的局部变量）创建的，所以第9行拷贝构造函数当然又被使用。现在，对于h2的标识符，名字变成了“ h拷贝的拷贝 ”。因为它是从拷贝拷过来的，这个拷贝是函数 f( ) 内部对象。在对象返回之后，函数结束之前，对象数暂时变为 3，但此后内部对象“ h拷贝 ”被销毁。在13行完成对f( )调用后，仅有2个对象h和h2 。这时我们可以看到h2最终是“ h拷贝的拷贝 ”。

• 临时对象

15行开始调用f(h)，这次调用忽略了返回值。在16行可以看到恰好在参数传入之前，拷贝构造函数被调用。和前面一样， 21行显示了为返回值而调用拷贝构造函数。但是，拷贝构造函数必须有一个作为它的目的地（this指针）的工作地址。对象返回到哪里？

每当编译器需要正确地计算一个表达式时，编译器可以创建一个临时对象。在这种情况下，编译器创建一个我们甚至看不见的对象作为函数 f( )忽略了的返回值的目的地地址。这个临时对象的生存期应尽可能地短，这样，空间就不会被这些等待被销毁且占珍贵资源的临时对象搞乱。在一些情况下，临时对象可能立即传递给另外的函数。但在现在这种情况下，在函数调用

之后，不需要临时对象，所以一旦函数调用以对内部对象调用析构函数（23和24行）的方式结束，临时对象就被销毁（25和26行）。

在28-31行，对象h2被销毁了，接着对象h被销毁。对象记数非常正确地回到了0。

10.3.3 缺省拷贝构造函数

因为拷贝构造函数实现传值方式的参数传递和返回，所以在这种简单结构情况下，编译器将有效地创建一个缺省拷贝构造函数，这非常重要。在C中也是这样。然而，直到目前所看到的一切都是缺省的原始行为：位拷贝（bitcopy）。

当包括更复杂的类型时，如果没有创建拷贝构造函数，C++编译器也将自动地为我们创建拷贝构造函数。这是因为在这里用位拷贝是没有意义的，它并不能达到我们的目的。

这儿有一个例子显示编译器采取的更具智能的方法。设想我们创建了一个包括几个现有类的对象的新类。这个创建类的方法被称为组合（composition），它是从现有类创建新类的方法之一。现在，假设我们用这个方法快速创建一个新类来解决某个问题。因为我们还不知道拷贝构造函数，所以没有创建它。下面的例子演示了当编译器为我们的新类创建缺省拷贝构造函数时编译器干了那些事。

```
//: AUTOCC.CPP -- Automatic copy-constructor
#include <iostream.h>
#include <string.h>

class withCC { // With copy-constructor
public:
  // Explicit default constructor required:
  withCC() {}
  withCC(const withCC&) {
    cout << "withCC(withCC&)" << endl;
  }
};

class woCC { // Without copy-constructor
  enum { bsz = 30 };
  char buf[bsz];
public:
  woCC(const char* msg = 0 ) {
    memset(buf, 0, bsz);
    if(msg) strncpy(buf, msg, bsz);
  }
  void print(const char* msg = 0) const {
    if(msg) cout << msg << ": ";
    cout << buf << endl;
  }
};

class composite {
  withCC WITHCC; // Embedded objects
```

```
  woCC WOCC;
public:
  composite() : WOCC("composite()") {}
  void print(const char* msg = 0) {
    WOCC.print(msg);
  }
};

main() {
  composite c;
  c.print("contents of c");
  cout << "calling composite copy-constructor"
       << endl;
  composite c2 = c;  // Calls copy-constructor
  c2.print("contents of c2");
}
```

类withCC有一个拷贝构造函数，这个函数只是简单地宣布它被调用。在类 composite中，使用缺省的构造函数创建一个 withCC类的对象。如果在类 withCC中根本没有构造函数，编译器将自动地创建一个缺省的构造函数。不过在这种情况下，这个构造函数什么也不做。然而，如果我们加了一个拷贝构造函数，我们就告诉了编译器我们将自己处理构造函数的创建，编译器将不再为我们创建缺省的构造函数。并且除非我们显式地创建一个缺省的构造函数，就如同为类withCC所做的那样，否则，编译器会指示出错。

类woCC没有拷贝构造函数，但它的构造函数将在内部缓冲器存储一个信息，这个信息可以使用print( )函数打印出来。这个构造函数在类composite构造函数的初始化表达式表（初始化表达式表已在第7章简单地介绍过了，并将在第 13章中全面介绍）中被显式地调用。这样做的原因在以后将会明白。

类composite既含有 withCC类的成员对象又含有 woCC类的成员对象（注意内嵌的对象WOCC在构造函数初始化表达式表中被初始化）。类composite没有显式地定义拷贝构造函数。然而，在main( )函数中，按下面的定义使用拷贝构造函数创建了一个对象。

```
composite c2 = c ;
```

类composite的拷贝构造函数由编译器自动创建，程序的输出显示了它是如何被创建的。

为了对使用组合（和继承的方法，将在第 13章介绍）的类创建拷贝构造函数，编译器递归地为所有的成员对象和基本类调用拷贝构造函数。如果成员对象也含有别的对象，那么后者的拷贝构造函数也将被调用。所以，在这里，编译器也为类 withCC调用拷贝构造函数。程序的输出显示了这个构造函数被调用。因为 woCC没有拷贝构造函数，编译器为它创建一个，它是缺省的位拷贝（bitcopy）的行为，编译器在类 composite的拷贝构造函数内部调用这个缺省的拷贝构造函数，于是在main中调用的composite::print( )显示的c2.WOCC的内容与c.WOCC内容将是相同的。编译器获得一个拷贝构造函数的过程被称为 memberwise initialization。

最好的方法是创建自己的拷贝构造函数而不让编译器创建。这样就能保证程序在我们的控制之下。

10.3.4 拷贝构造函数方法的选择

现在，我们可能已头晕了。我们可能想，怎样才能不必了解拷贝构造函数就能写一个具有

一定功能的类。但是我们别忘了：仅当准备用传值的方式传递类对象时，才需要拷贝构造函数。如果不需要这么做，就不要拷贝构造函数。

1. 防止传值方式传递

我们也许会说：“如果我自己不写拷贝构造函数，编译器将为我创建。所以，我怎么能保证一个对象永远不会被通过传值方式传递呢？”

有一个简单的技术防止通过传值方式传递：声明一个私有（private）拷贝构造函数。我们甚至不必去定义它，除非我们的成员函数或友元（friend）函数需要执行传值方式的传递。如果用户试图用传值方式传递或返回对象，编译器将会发出一个出错信息。这是因为拷贝构造函数是私有的。因为我们已显式地声明我们接管了这项工作，所以编译器不再创建缺省的拷贝构造函数。

这儿提供了一个例子：

```
//: STOPCC.CPP -- Preventing copy-construction

class noCC {
  int i;
  noCC(const noCC&); // No definition
public:
  noCC(int I = 0) : i(I) {}
};

void f(noCC);

main() {
  noCC n;
//! f(n); // Error: copy-constructor called
//! noCC n2 = n; // Error: c-c called
//! noCC n3(n); // Error: c-c called
}
```

注意使用更普通的形式

```
noCC(const noCC&);
```

这里使用了const。

2. 改变外部对象的函数

一般来讲，引用语法比指针语法更好，然而对于读者来说，它使得意思变得模糊。例如，在iostreams库函数中，一个重载版函数get( )是用一个char&作为参数，函数通过插入get( )的结果而改变它的参数。然而，当我们阅读使用这个函数的代码时，我们不会立即明白外面的对象正被改变：

```
char c;
cin.get(c);
```

事实上函数调用看起来像一个传值传递，暗示着外部对象没有被改变。

正因为如此，当传递一个可被修改的参数时，从代码维护的观点看，使用指针可能安全些。所以除非我们打算通过地址修改外部对象（这个地址通过非const指针传递），要不然都用const引用传递地址。因为这样，读者更容易读懂我们的代码。

10.4 指向成员的指针（简称成员指针）

指针是指向一些内存地址的变量，既可以是数据的地址也可以是函数的地址。所以，可以在运行时改变指针指向的内容。除了C++的成员指针(pointer-to-member)选择的内容是在类之外，C++的成员指针遵从同样的原则。困难的是所有的指针需要一个地址，但在类内部没有地址；选择一个类的成员意味着在类中偏移。只有把这个偏移和具体对象的开始地址结合，才能得到实际地址。成员指针的语法要求选择一个对象的同时逆向引用成员指针。

为了理解这个语法，先来考虑一个简单的结构：

```
struct simple { int a ; };
```

如果有一个这个结构的指针sp和对象so，可以通过下面方法选择成员：

```
sp->a ;
so.a ;
```

现在，假设有一个普通的指向integer的指针ip。为了取得ip指向的内容，用一个*号逆向引用指针的引用。

```
*ip = 4 ;
```

最后，考虑如果有一个指向一个类对象成员，甚至假设它代表对象内一定的偏移，将会发生什么？为了取得指针指向的内容，必须用 *号逆向引用。但是，它只是一个对象内的偏移，所以还必须要指定那个对象。因此， *号要和逆向引用的对象结合。像下面使用 simple类的例子：

```
sp->*pm = 47 ;
so.*pm = 47 ;
```

所以，对于指向一个对象的指针新的语法变为->*，对于一个对象或引用则为.*。现在，让我们看看定义pm的语法是什么？其实它像任何一个指针，必须说出它指向什么类型。并且，在定义中也要使用一个‘ * ’号。唯一的区别只是必须说出这个成员指针使用什么类的对象。当然，这是用类名和全局操作符实现的。因此，可表示如下：

```
int simple::*pm ;
```

当我们定义它时（或任何别的时间），也可以初始化成员指针：

```
int simple::*pm = &simple::a ;
```

因为引用到一个类而非那个类的对象，所以没有 simple::a的确切“地址”。因而，&simple::a仅可作为成员指针的语法表示。

函数

这里提供一个为成员函数产生成员指针（pointer-to-member）的例子。指向函数的指针定义像下面的形式：

```
int (*fp)(float) ;
```

(*fp)的圆括号用来迫使编译器正确判断定义。没有圆括号，这个表达式就是一个返回 int*值的函数。

为了定义和使用一个成员函数的指针，圆括号扮演同样重要的角色。假设在一个结构内有一个函数：

```
struct simple2 { int f(float); } ;
```

通过给普通函数插入类名和全局操作符就可以定义一个指向成员函数的指针：

```
int (simple2::*fp) (float) ;
```

当创建它时或其他任何时候，可以对它初始化：

```
int (simple2::*fp) (float) = &simple2::f ;
```

和其他普通函数一样，&号是可选的；可以用不带参数表的函数标识符来表示地址：

```
fp = simple2::f ;
```

• 一个例子

在程序运行时，我们可以改变指针所指的内容。因此在运行时我们就可以通过指针选择来改变我们的行为，这就为程序设计提供了重要的灵活性。成员指针也一样，它允许在运行时选择一个成员。特别的，当我们的类只有公有 (public)成员函数（数据成员通常被认为是内部实现的一部分）时，就可以用指针在运行时选择成员函数，下面的例子正是这样：

```
//: PMEM.CPP -- Pointers to members

class widget {
public:
  void f(int);
  void g(int);
  void h(int);
  void i(int);
};

void widget::h(int) {}

main() {
  widget w;
  widget* wp = &w;
  void (widget::*pmem)(int) = &widget::h;
  (w.*pmem)(1);
  (wp->*pmem)(2);
}
```

当然，期望一般用户创建如此复杂的表达式不是特别合乎情理。如果用户必须直接操作成员指针，那么typedef是适合的。为了安排得当，可以使用成员指针作为内部执行机制的一部分。这儿提供一个前述的在类内使用成员指针的例子。用户所要做的是传递一个数字以选择一个函数。[1]

```
//: PMEM2.CPP -- Pointers to members
#include <iostream.h>
class widget {
  void f(int) const {cout << "widget::f()\n";}
  void g(int) const {cout << "widget::g()\n";}
  void h(int) const {cout << "widget::h()\n";}
  void i(int) const {cout << "widget::i()\n";}
```

[1] 感谢Owen Mortensen提供这个例子。

```
    enum { count = 4 };
    void (widget::*fptr[count])(int) const;
  public:
    widget() {
      fptr[0] = &widget::f; // Full spec required
      fptr[1] = &widget::g;
      fptr[2] = &widget::h;
      fptr[3] = &widget::i;
    }
    void select(int I, int J) {
      if(I < 0 || I >= count) return;
      (this->*fptr[I])(J);
    }
    int Count() { return count; }
  };

  main() {
    widget w;
    for(int i = 0; i < w.Count(); i++)
      w.select(i, 47);
  }
```

在类接口和main( )函数里，可以看到，包括函数本身在内的整个实现被隐藏了。代码甚至必须请求Count( )函数。用这个方法，类的实现者可以改变大量函数而不影响使用这个类的代码。

在构造函数中，成员指针的初始化似乎被过分地指定了。是否可以这样写：

```
fptr[1] = &g ;
```

因为名字g在成员函数中出现，是否可以自动地认为在这个类范围内呢？问题是这不符合成员函数的语法，它的语法要求每个人，尤其编译器，能够判断将要进行什么。相似地，当成员函数被逆向引用时，它看起来像这样：

```
(this->*fptr[i]) (j);
```

它仍被过分地指定了，this似乎多余。正如前面所讲的，当它被逆向引用时，语法也需要成员指针总是和一个对象绑定在一起。

10.5 小结

C++的指针和C中的指针是非常相似的，这是非常好的。否则，许多C代码在C++中将不会被正确地编译。仅在出现危险赋值的地方，编译器会产生出错信息。假设我们确实想这样赋值，编译器的出错可以用简单的（和显式的！）映射（cast）清除。

C++还从Algol和Pascal中引进引用(reference)概念，引用就像一个能自动被编译器逆向引用的常量指针一样。引用占一个地址，但我们可以把它看成一个对象。引用是操作符重载语法（下一章的主题）的重点，它也为普通函数值传递和返回对象增加了语法的便利。

拷贝构造函数采用相同类型的对象引用作为它的参数，它可以被用来从现有的类创建新类。当用传值方式传递或返回一个对象时，编译器自动调用这个拷贝构造函数。虽然，编译器将自

动地创建一个拷贝构造函数，但是，如果认为需要为我们的类创建一个拷贝构造函数，应该自己定义它以确保正确的操作。如果不想通过传值方式传递和返回对象，应该创建一个私有的(private)拷贝构造函数。

成员指针和普通指针一样具有相同的功能：可以在运行时选取特定存储单元(数据或函数)。成员指针只和类成员一起工作而不和全局数据或函数一起工作。通过使用成员指针，我们的程序设计可以在运行时灵活地改变。

10.6 练习

1. 写一个函数，这个函数用一个char&作参数并且修改该参数。在main( )函数里，打印一个char变量，使用这个变量做参数，调用我们设计的函数。然后，再次打印此变量以证明它已被改变。这样做是如何影响程序的可读性的？

2. 写一个有拷贝构造函数的类，在拷贝构造函数里用 cout自我声明。现在，写一个函数，这个函数通过传值方式传入我们新类的对象。写另一个函数，在这个函数内创建这个新类的局部对象，通过传值方式返回这个对象。调用这些函数以证明通过传值方式传递和返回对象时，拷贝构造函数确实悄悄地被调用了。

3. 努力发现如何使得我们的编译器产生汇编语言，并请为PASSTRUC.CPP产生汇编代码。跟踪和揭示我们的编译器为传递和返回大结构产生代码的方法。

第11章　运算符重载

运算符重载只是一种“语法修饰”，这意味着它是另一种调用函数的方法。

不同之处是对于函数的参数不是出现在圆括号内，而是在我们总认为是运算符的字符的附近。

但在C++中，可以定义一个和类一起工作的新运算符。除了这个名字函数以关键字operator开始，以运算符本身结束以外，这个定义和一个普通函数是一样的。这是仅有的差别。它像其他函数一样也是一个函数，当编译器看到它以适当的模式出现时，就调用它。

11.1　警告和确信

对于运算符重载，人们容易变得过于热心。首先，它是一个娱乐玩具。注意，它仅仅是一个语法修饰，是另外一种调用函数的方法而已。用这种眼光看，没有理由重载一个运算符，除非它会使包含我们的类的代码变得更易写，尤其是更易读。（记住，读代码的情况更多）如果不是这种情况，就不必麻烦去重载运算符。

对于运算符重载，另外一个通常的反映是恐慌：突然，C运算符不再有熟悉的意思。“所有的东西都改变了，我的所有C代码将做不同的事情！”但这不是事实。所有用于仅包含内部数据类型的表达式的运算符是不可能被改变的。我们永远不能重载下面的运算符使执行的行为不同。

```
1 << 4;
```

或者重载运算符使得下面的表达式有意义。

```
1.414 << 2;
```

仅仅是包含用户自定义类型的表达式可以有重载的运算符。

11.2　语法

定义一个重载运算符就像定义一个函数，只是该函数的名字是operator@，这里@代表运算符。函数参数表中参数的个数取决于两个因素：

1) 运算符是一元的（一个参数）还是二元的（两个参数）。

2) 运算符被定义为全局函数（对于一元是一个参数，对于二元是两个参数）还是成员函数（对于一元没有参数，对于二元是一个参数 — 对象变为左侧参数）。

这里有一个很小的类来显示运算符重载语法。

```
//: OPOVER.CPP -- Operator overloading syntax
#include <iostream.h>

class integer {
  int i;
public:
  integer(int I) { i = I; }
  const integer
```

```
  operator+(const integer& rv) const {
    cout << "operator+" << endl;
    return integer(i + rv.i);
  }
  integer&
  operator+=(const integer& rv){
    cout << "operator+=" << endl;
    i += rv.i;
    return *this;
  }
};

main() {
  cout << "built-in types:" << endl;
  int i = 1, j = 2, k = 3;
  k += i + j;
  cout << "user-defined types:" << endl;
  integer I(1), J(2), K(3);
  K += I + J;
}
```

这两个重载的运算符被定义为内联成员函数。对于二元运算符，单个参数是出现在运算符右侧的那个。当一元运算符被定义为成员函数时，没有参数。成员函数被运算符左侧的对象调用。

对于非条件运算符（条件运算符通常返回一个布尔值），如果两个参数是相同的类型，希望返回和运算相同类型的对象或引用。如果它们不是相同类型，它作什么样的解释就取决于程序设计者。用这种方法可以组合复杂的表达式：

```
K += I + J;
```

运算符+号产生一个新的整数（临时的），这个整数被用作运算符‘ += ’的rv参数。一旦这个临时整数不再需要时就被消除。

11.3 可重载的运算符

虽然可以重载几乎所有C中可用的运算符，但使用它们是相当受限制的。特别地，不能结合C中当前没有意义的运算符（例如 **求幂），不能改变运算符的优先级，不能改变运算符的参数个数。这样限制有意义 —— 所有这些行为产生的运算符只会造成意思混淆而不是使之清楚。

下面两个小部分给出所有“ 经常用的 ”运算符的例子，这些被重载的运算符的形式会经常用到。

11.3.1 一元运算符

下面的例子显示了所有一元运算符重载的语法，它们既以全局函数形式又以成员函数形式表示。它们将扩充先前显示的类 integer和加入新类byte。具体运算符的意思取决于如何使用它们。

```
//: UNARY.CPP -- Overloading unary operators
#include <iostream.h>
class integer {
  long i;
  integer* This() { return this; }
public:
  integer(long I = 0) : i(I) {}
  // No side effects takes const& argument:
  friend const integer&
    operator+(const integer& a);
  friend const integer
    operator-(const integer& a);
  friend const integer
    operator~(const integer& a);
  friend integer*
    operator&(integer& a);
  friend int
    operator!(const integer& a);
  // Side effects don't take const& argument:
  // Prefix:
  friend const integer&
    operator++(integer& a);
  // Postfix:
  friend const integer
    operator++(integer& a, int);
  // Prefix:
  friend const integer&
    operator--(integer& a);
  // Postfix:
  friend const integer
    operator--(integer& a, int);
};

// Global operators:
const integer& operator+(const integer& a) {
  cout << "+integer\n";
  return a; // Unary + has no effect
}
const integer operator-(const integer& a) {
  cout << "-integer\n";
  return integer(-a.i);
}
const integer operator~(const integer& a) {
  cout << "~integer\n";
  return integer(~a.i);
}
```

```
integer* operator&(integer& a) {
  cout << "&integer\n";
  return a.This(); // &a is recursive!
}
int operator!(const integer& a) {
  cout << "!integer\n";
  return !a.i;
}
// Prefix; return incremented value
const integer& operator++(integer& a) {
  cout << "++integer\n";
  a.i++;
  return a;
}
// Postfix; return the value before increment:
const integer operator++(integer& a, int) {
  cout << "integer++\n";
  integer r(a.i);
  a.i++;
  return r;
}
// Prefix; return decremented value
const integer& operator--(integer& a) {
  cout << "--integer\n";
  a.i--;
  return a;
}
// Postfix; return the value before decrement:
const integer operator--(integer& a, int) {
  cout << "integer--\n";
  integer r(a.i);
  a.i--;
  return r;
}

void f(integer a) {
  +a;
  -a;
  ~a;
  integer* ip = &a;
  !a;
  ++a;
  a++;
  --a;
  a--;
}
```

```
// Member operators (implicit "this"):
class byte {
  unsigned char b;
public:
  byte(unsigned char B = 0) : b(B) {}
  // No side effects: const member function:
  const byte& operator+() const {
    cout << "+byte\n";
    return *this;
  }
  const byte operator-() const {
    cout << "-byte\n";
  return byte(-b);
}
const byte operator~() const {
  cout << "~byte\n";
  return byte(~b);
}
byte operator!() const {
  cout << "!byte\n";
  return byte(!b);
}
byte* operator&() {
  cout << "&byte\n";
  return this;
}
// Side effects: non-const member function:
const byte& operator++() { // Prefix
  cout << "++byte\n";
  b++;
  return *this;
}
const byte operator++(int) { // Postfix
  cout << "byte++\n";
  byte before(b);
  b++;
  return before;
}
const byte& operator--() { // Prefix
  cout << "--byte\n";
  --b;
  return *this;
}
const byte operator--(int) { // Postfix
  cout << "byte--\n";
```

```
  byte before(b);
  --b;
  return before;
}
};

void g(byte b) {
  +b;
  -b;
  ~b;
  byte* bp = &b;
  !b;
  ++b;
  b++;
  --b;
  b--;
}

main() {
  integer a;
  f(a);
  byte b;
  g(b);
}
```

根据参数传递的方法，将函数分组。如何传递和返回参数的方针在后面给出。上面的形式（和下一小节的形式）是典型的使用形式，所以当重载自己的运算符时可以以它们作为范式开始。

• 自增和自减

重载的++和--号运算符出现了两难选择的局面，这是因为希望根据它们出现在它们作用的对象前面（前缀）还是后面（后缀）来调用不同的函数。解决是很简单的，但一些人在开始时却发现它们容易令人混淆。例如当编译器看到 ++a（先自增）时，它就调用 operator++(a);但当编译器看到a++时，它就调用operator++(a,int)。即编译器通过调用不同的函数区别这两种形式。在UNARY.CPP成员函数版中，如果编译器看到 ++b，它就产生一个对B::perator++( )的调用；如果编译器看到b++，它就产生一个对B::operator++(int)的调用。

除非对于前缀和后缀版本用不同的函数调用，否则用户永远看不到它动作的结果。然而，实质上两个函数调用有不同的署名，所以它们和两个不同函数体相连。编译器为 int参数（因为这个值永远不被使用，所以它永远不会被赋给一个标识符）传递一个哑元常量值用来为后缀版产生不同的署名。

11.3.2 二元运算符

下面的清单是用二元运算符重复 UNARY.CPP的例子。全局版本和成员函数版本都在里面。

```
//: BINARY.CPP -- Overloading binary operators
#include <fstream.h>
#include "..\allege.h"
```

```
ofstream out("binary.out");

class integer { // Combine this with UNARY.CPP
  long i;
public:
  integer(long I = 0) : i(I) {}
  // Operators that create new, modified value:
  friend const integer
    operator+(const integer& left,
              const integer& right);
  friend const integer
    operator-(const integer& left,
              const integer& right);
  friend const integer
    operator*(const integer& left,
              const integer& right);
  friend const integer
    operator/(const integer& left,
              const integer& right);
friend const integer
  operator%(const integer& left,
            const integer& right);
friend const integer
  operator^(const integer& left,
            const integer& right);
friend const integer
  operator&(const integer& left,
            const integer& right);
friend const integer
  operator|(const integer& left,
            const integer& right);
friend const integer
  operator<<(const integer& left,
             const integer& right);
friend const integer
  operator>>(const integer& left,
             const integer& right);
// Assignments modify & return lvalue:
friend integer&
  operator+=(integer& left,
             const integer& right);
friend integer&
  operator-=(integer& left,
             const integer& right);
friend integer&
  operator*=(integer& left,
```

```
                const integer& right);
  friend integer&
    operator/=(integer& left,
                const integer& right);
  friend integer&
    operator%=(integer& left,
                const integer& right);
  friend integer&
    operator^=(integer& left,
               const integer& right);
  friend integer&
    operator&=(integer& left,
               const integer& right);
  friend integer&
    operator|=(integer& left,
               const integer& right);
  friend integer&
    operator>>=(integer& left,
                const integer& right);
  friend integer&
    operator<<=(integer& left,
                const integer& right);
  // Conditional operators return true/false:
  friend int
    operator==(const integer& left,
               const integer& right);
  friend int
    operator!=(const integer& left,
               const integer& right);
  friend int
    operator<(const integer& left,
              const integer& right);
  friend int
    operator>(const integer& left,
              const integer& right);
  friend int
    operator<=(const integer& left,
               const integer& right);
  friend int
    operator>=(const integer& left,
               const integer& right);
  friend int
    operator&&(const integer& left,
               const integer& right);
  friend int
    operator||(const integer& left,
```

```
                const integer& right);
  // Write the contents to an ostream:
  void print(ostream& os) const { os << i; }
};

const integer
  operator+(const integer& left,
            const integer& right) {
  return integer(left.i + right.i);
}
const integer
  operator-(const integer& left,
            const integer& right) {
  return integer(left.i - right.i);
}
const integer
  operator*(const integer& left,
            const integer& right) {
  return integer(left.i * right.i);
}
const integer
  operator/(const integer& left,
            const integer& right) {
  allege(right.i != 0, "divide by zero");
  return integer(left.i / right.i);
}
const integer
  operator%(const integer& left,
            const integer& right) {
  allege(right.i != 0, "modulo by zero");
  return integer(left.i % right.i);
}
const integer
  operator^(const integer& left,
            const integer& right) {
  return integer(left.i ^ right.i);
}
 const integer
   operator&(const integer& left,
             const integer& right) {
   return integer(left.i & right.i);
 }
 const integer
   operator|(const integer& left,
             const integer& right) {
```

```
  return integer(left.i | right.i);
}
const integer
  operator<<(const integer& left,
             const integer& right) {
  return integer(left.i << right.i);
}
const integer
  operator>>(const integer& left,
             const integer& right) {
  return integer(left.i >> right.i);
}
// Assignments modify & return lvalue:
integer& operator+=(integer& left,
                    const integer& right) {
   if(&left == &right) {/* self-assignment */}
   left.i += right.i;
   return left;
}
integer& operator-=(integer& left,
                    const integer& right) {
   if(&left == &right) {/* self-assignment */}
   left.i -= right.i;
   return left;
}
integer& operator*=(integer& left,
                    const integer& right) {
   if(&left == &right) {/* self-assignment */}
   left.i *= right.i;
   return left;
}
integer& operator/=(integer& left,
                    const integer& right) {
   allege(right.i != 0, "divide by zero");
   if(&left == &right) {/* self-assignment */}
   left.i /= right.i;
   return left;
}
integer& operator%=(integer& left,
                    const integer& right) {
   allege(right.i != 0, "modulo by zero");
   if(&left == &right) {/* self-assignment */}
   left.i %= right.i;
   return left;
}
integer& operator^=(integer& left,
```

```
                    const integer& right) {
  if(&left == &right) {/* self-assignment */}
  left.i ^= right.i;
  return left;
}
integer& operator&=(integer& left,
                    const integer& right) {
  if(&left == &right) {/* self-assignment */}
  left.i &= right.i;
  return left;
}
integer& operator|=(integer& left,
                    const integer& right) {
  if(&left == &right) {/* self-assignment */}
  left.i |= right.i;
  return left;
}
integer& operator>>=(integer& left,
                     const integer& right) {
  if(&left == &right) {/* self-assignment */}
  left.i >>= right.i;
  return left;
}
integer& operator<<=(integer& left,
                     const integer& right) {
  if(&left == &right) {/* self-assignment */}
  left.i <<= right.i;
  return left;
}
// Conditional operators return true/false:
int operator==(const integer& left,
               const integer& right) {
    return left.i == right.i;
}
int operator!=(const integer& left,
               const integer& right) {
    return left.i != right.i;
}
int operator<(const integer& left,
              const integer& right) {
    return left.i < right.i;
}
int operator>(const integer& left,
              const integer& right) {
    return left.i > right.i;
}
```

```
int operator<=(const integer& left,
               const integer& right) {
    return left.i <= right.i;
}
int operator>=(const integer& left,
               const integer& right) {
    return left.i >= right.i;
}
int operator&&(const integer& left,
               const integer& right) {
    return left.i && right.i;
}
int operator||(const integer& left,
               const integer& right) {
    return left.i || right.i;
}

void h(integer& c1, integer& c2) {
  // A complex expression:
  c1 += c1 * c2 + c2 % c1;
  #define TRY(op) \
  out << "c1 = "; c1.print(out); \
  out << ", c2 = "; c2.print(out); \
  out << ";  c1 " #op " c2 produces ";\
  (c1 op c2).print(out); \
  out << endl;
  TRY(+) TRY(-) TRY(*) TRY(/)
  TRY(%) TRY(^) TRY(&) TRY(|)
  TRY(<<) TRY(>>) TRY(+=) TRY(-=)
  TRY(*=) TRY(/=) TRY(%=) TRY(^=)
  TRY(&=) TRY(|=) TRY(>>=) TRY(<<=)
  // Conditionals:
  #define TRYC(op) \
  out << "c1 = "; c1.print(out); \
  out << ", c2 = "; c2.print(out); \
  out << ";  c1 " #op " c2 produces ";\
  out << (c1 op c2); \
  out << endl;
  TRYC(<) TRYC(>) TRYC(==) TRYC(!=) TRYC(<=)
  TRYC(>=) TRYC(&&) TRYC(||)
}

// Member operators (implicit "this"):
class byte { // Combine this with UNARY.CPP
  unsigned char b;
```

```
public:
  byte(unsigned char B = 0) : b(B) {}
  // No side effects: const member function:
 const byte
   operator+(const byte& right) const {
   return byte(b + right.b);
 }
 const byte
   operator-(const byte& right) const {
   return byte(b - right.b);
 }
 const byte
   operator*(const byte& right) const {
   return byte(b * right.b);
 }
 const byte
   operator/(const byte& right) const {
   allege(right.b != 0, "divide by zero");
   return byte(b / right.b);
 }
 const byte
   operator%(const byte& right) const {
   allege(right.b != 0, "modulo by zero");
   return byte(b % right.b);
 }
 const byte
   operator^(const byte& right) const {
   return byte(b ^ right.b);
 }
 const byte
   operator&(const byte& right) const {
   return byte(b & right.b);
 }
 const byte
   operator|(const byte& right) const {
   return byte(b | right.b);
 }
 const byte
   operator<<(const byte& right) const {
   return byte(b << right.b);
 }
const byte
  operator>>(const byte& right) const {
  return byte(b >> right.b);
}
```

```
// Assignments modify & return lvalue.
// operator= can only be a member function:
byte& operator=(const byte& right) {
  // Handle self-assignment:
  if(this == &right) return *this;
  b = right.b;
  return *this;
}
byte& operator+=(const byte& right) {
  if(this == &right) {/* self-assignment */}
  b += right.b;
  return *this;
}
byte& operator-=(const byte& right) {
  if(this == &right) {/* self-assignment */}
  b -= right.b;
  return *this;
}
byte& operator*=(const byte& right) {
  if(this == &right) {/* self-assignment */}
  b *= right.b;
  return *this;
}
byte& operator/=(const byte& right) {
  allege(right.b != 0, "divide by zero");
  if(this == &right) {/* self-assignment */}
  b /= right.b;
  return *this;
}
byte& operator%=(const byte& right) {
  allege(right.b != 0, "modulo by zero");
  if(this == &right) {/* self-assignment */}
  b %= right.b;
  return *this;
}
byte& operator^=(const byte& right) {
  if(this == &right) {/* self-assignment */}
  b ^= right.b;
  return *this;
}
byte& operator&=(const byte& right) {
  if(this == &right) {/* self-assignment */}
  b &= right.b;
  return *this;
}
byte& operator|=(const byte& right) {
```

```
    if(this == &right) {/* self-assignment */}
    b |= right.b;
    return *this;
  }
  byte& operator>>=(const byte& right) {
    if(this == &right) {/* self-assignment */}
    b >>= right.b;
    return *this;
  }
  byte& operator<<=(const byte& right) {
    if(this == &right) {/* self-assignment */}
    b <<= right.b;
    return *this;
  }
  // Conditional operators return true/false:
  int operator==(const byte& right) const {
      return b == right.b;
  }
  int operator!=(const byte& right) const {
      return b != right.b;
  }
  int operator<(const byte& right) const {
      return b < right.b;
  }
  int operator>(const byte& right) const {
      return b > right.b;
  }
  int operator<=(const byte& right) const {
      return b <= right.b;
  }
  int operator>=(const byte& right) const {
      return b >= right.b;
  }
  int operator&&(const byte& right) const {
      return b && right.b;
  }
  int operator||(const byte& right) const {
      return b || right.b;
  }
  // Write the contents to an ostream:
  void print(ostream& os) const {
    os << "0x" << hex << int(b) << dec;
  }
};

void k(byte& b1, byte& b2) {
```

```
  b1 = b1 * b2 + b2 % b1;

  #define TRY2(op) \
  out << "b1 = "; b1.print(out); \
  out << ", b2 = "; b2.print(out); \
  out << ";  b1 " #op " b2 produces ";\
  (b1 op b2).print(out); \
  out << endl;

  b1 = 9; b2 = 47;
  TRY2(+) TRY2(-) TRY2(*) TRY2(/)
  TRY2(%) TRY2(^) TRY2(&) TRY2(|)
  TRY2(<<) TRY2(>>) TRY2(+=) TRY2(-=)
  TRY2(*=) TRY2(/=) TRY2(%=) TRY2(^=)
  TRY2(&=) TRY2(|=) TRY2(>>=) TRY2(<<=)
  TRY2(=) // Assignment operator

  // Conditionals:
  #define TRYC2(op) \
  out << "b1 = "; b1.print(out); \
  out << ", b2 = "; b2.print(out); \
  out << ";  b1 " #op " b2 produces ";\
  out << (b1 op b2); \
  out << endl;

  b1 = 9; b2 = 47;
  TRYC2(<) TRYC2(>) TRYC2(==) TRYC2(!=) TRYC2(<=)
  TRYC2(>=) TRYC2(&&) TRYC2(||)

  // Chained assignment:
  byte b3 = 92;
  b1 = b2 = b3;
}

main() {
  integer c1(47), c2(9);
  h(c1, c2);
  out << "\n member functions:" << endl;
  byte b1(47), b2(9);
  k(b1, b2);
}
```

可以看到运算符‘ = ’仅允许作为成员函数。这将在后面解释。

作为总的方针，我们注意到在运算符重载中所有赋值运算符都有代码用于核对自赋值 (self-assignment)。在一些情况下，这是不需要的。例如，可以用运算符‘ += ’写A+=A，使得A自身相加。最重要的核对自赋值的地方是运算符‘ = ’，因为复杂的对象可能因为它而发生灾难性

的结果（在一些情况下这不会有问题，不管怎么说，在写运算符‘ = ’时，应该小心一些）。

先前的两个例子中的运算符重载处理单一类型。也可能重载运算符处理混合类型，所以可以“ 把苹果加到橙子里 ”。然而，在开始进行运算符重载之前，应该看一下本章后面有关自动类型转换部分。经常在正确的地方使用类型转换可以减少许多运算符重载。

11.3.3 参数和返回值

开始看UNARY.CPP和BINARY.CPP例子时会发现参数传递和返回方法完全不同，这似乎有点令人混淆。虽然可以用任何想用的方法传递和返回参数，但这些例子方法却不是随便选择的。它们遵守一种非常合乎逻辑的模式，我们在大部分情况下都应选择这种模式：

1) 对于任何函数参数，如果仅需要从参数中读而不改变它，缺省地应当按 const引用来传递它。普通算术运算符（像+和-号等）和布尔运算符不会改变参数，所以以const引用传递是使用的主要方式。当函数是一个类成员的时候，就转换为 const成员函数。只是对于会改变左侧参数的赋值运算符(operator-assignment，像+=）和运算符‘ = ’，左侧参数才不是常量(constant)，但因为参数将被改变，所以参数仍然按地址传递。

2) 应该选择的返回值取决于运算符所期望的类型。（可以对参数和返回值做任何想做的事）如果运算符的效果是产生一个新值，将需要产生一个作为返回值的新对象。例如，integer::operator+必须生成一个操作数之和的 integer对象。这个对象作为一个 const通过传值方式返回，所以作为一个左值结果不会被改变。

3) 所有赋值运算符改变左值。为了使得赋值结果用于链式表达式（像 A=B=C），应该能够返回一个刚刚改变了的左值的引用。但这个引用应该是 const还是nonconst呢？虽然我们是从左向右读表达式 A=B=C，但编译器是从右向左分析这个表达式，所以并非一定要返回一个 nonconst值来支持链式赋值。然而人们有时希望能够对刚刚赋值的对象进行运算，例如（A=B）.foo（ ），这是B赋值给A后调用foo( )。因此所有赋值运算符的返回值对于左值应该是nonconst引用。

4) 对于逻辑运算符，人们希望至少得到一个 int返回值，最好是bool返回值。（在大多数编译器支持C++内置bool类型之前开发的库函数使用int或typedef等价物）。

5) 因为有前缀和后缀版本，所以自增和自减运算符出现了两难局面。两个版本都改变对象，所以不能把这个对象看作一个 const。因此，前缀版本返回这个对象被改变后的值。这样，用前缀版本我们只需返回*this作为一个引用。因为后缀版本返回改变之前的值，所以被迫创建一个代表这个值的单个对象并返回它。因此，如果想保持我们的本意，对于后缀必须通过传值方式返回。（注意，我们经常会发现自增和自减运算返回一个 int 值或bool值，例如用来指示是否有一个循环子(iterator)在表的结尾）。现在问题是：这些应该按const被返回还是按nonconst被返回？如果允许对象被改变，一些人写了表达式(++A).foo( )，则foo( )作用在A上。但对于表达式(A++).foo( )，foo( )作用在通过后缀运算符 ++号返回的临时对象上。临时对象自动定为 const，所以被编译器标记。但为了一致性，使两者都是 const更有意义，就像这儿所做的。因为想给自增和自减运算符赋予各种意思，所以它们需要就事论事考虑。

1. 按const通过传值方式返回

按const通过传值方式返回，开始看起来有些微妙，所以值得多加解释。我们来考虑二元运算符+号。假设在一个表达式像 f(A+B)中使用它，A+B的结果变为一个临时对象，这个对象用于f( ) 调用。因为它是临时的，自动被定为const，所以无论使返回值为const还是不这样做都没有影响。

然而，也可能发送一个消息给A+B的返回值而不是仅传递给一个函数。例如，可以写表达式(A+B).g( )，这里g( )是integer的成员函数。通过设返回值为 const，规定了对于返回值只有const成员函数才可以被调用。用 const是恰当的，这是因为这样可以防止在很可能丢失的对象中存贮有价值的信息。

2. 返回效率

当为通过传值方式返回而创建一个新对象时，要注意使用的形式。例如用运算符 +号：

```
return integer (left.i + right.i) ;
```

一开始看起来像是一个“ 对一个构造函数的调用 ”，但其实并非如此。这是临时对象语法，它是这样陈述的：“ 创建一个临时对象并返回它 ”。因为这个原因，我们可能认为如果创建一个命名的本地对象并返回它结果将会是一样的。其实不然。如果像下面这样表示，将发生三件事。首先，tmp对象被创建，与此同时它的构造函数被调用。然后，拷贝构造函数把 tmp拷贝到返回值外部存储单元里。最后，当tmp在作用域的结尾时调用析构函数。

```
integer tmp(left.i + right.i) ;
return tmp ;
```

相反，“ 返回临时对象 ”的方法是完全不同的。看这样情况时，编译器明白对创建的对象没有其他需求，只是返回它，所以编译器直接地把这个对象创建在返回值外面的内存单元。因为不是真正创建一个局部对象，所以仅需要单个的普通构造函数调用（不需要拷贝构造函数），并且不会调用析构函数。因此，这种方法不需要什么花费，效率是非常高的。

11.3.4 与众不同的运算符

有些其他的运算符对于重载语法有明显的不同。

下标运算符‘ [ ] ’必须是成员函数并且它需要单个参数。因为它暗示对象像数组一样动作，可以经常从这个运算符返回一个引用，所以它可以被很方便地用于等号左侧。这个运算符经常被重载；可以在本书其他部分看到有关的例子。

当逗号出现在逗号运算对象左右时，逗号运算符被调用。然而，逗号运算符在函数参数表中出现时不被调用，此时，逗号仅在对象中起分割作用。除了使语言保持一致性外，这个运算符似乎没有许多实际用途。这儿有一个例子用于显示当逗号出现在对象前面以及后面时，逗号函数是如何被调用的：

```
//: COMMA.CPP -- Overloading operator,
#include <iostream.h>

class after {
public:
  const after& operator,(const after&) const {
    cout << "after::operator,()" << endl;
    return *this;
  }
};

class before {};

before& operator,(int, before& b) {
```

```
  cout << "before::operator,()" << endl;
  return b;
}

main() {
  after a, b;
  a, b;  // Operator comma called

  before c;
  1, c;  // Operator comma called
}
```

全局函数允许逗号放在被讨论的对象的前面。这里的用法显得相当晦涩和令人怀疑。虽然，可以用逗号分割作为表达式的一部分，但它太敏感以至于在大多数情况下不能使用。

运算符( )的函数调用必须是成员函数，它是唯一的允许在它里面有任意个参数的函数。这使得对象看起来像一个真正的函数名，因此，它最好为仅有单一运算的类型使用，或至少是特别优先的一种类型。

运算符new和delete控制动态内存分配，也可被重载。这是下一章非常重要的主题。

运算符->*是其行为像所有其他二元运算符的二元运算符。它是为模仿前一章介绍的内部数据类型的成员指针行为的情形而提供的。

为了使一个对象的表现是一个指针，就要设计使用灵巧 (smart)指针：->。如果想为类包装一个指针以使得这个指针安全，或是在一个普通的循环子（iterator）的用法中，则这样做特别有用。循环子是一个对象，这个对象可以作用于其他对象的包容器或集合上，每次选择它们中的一个，而不用提供对包容器实现的直接访问。（在类函数里经常发现包容器和循环子。）

灵巧指针必须是成员函数。它有一个附加的非典型的内容：它必须返回一个对象（或对象的引用），这个对象也有一个灵巧指针或指针，可用于选择这个灵巧指针所指向的内容。这儿提供了一个例子：

```
//: SMARTP.CPP -- Smart pointer example
#include <iostream.h>
#include <string.h>

class obj {
  static int i, j;
public:
  void f() { cout << i++ << endl; }
  void g() { cout << j++ << endl; }
};

// Static member definitions:
int obj::i = 47;
int obj::j = 11;

// Container:
class obj_container {
  enum { sz = 100 };
```

```
  obj* a[sz];
  int index;
public:
  obj_container() {
    index = 0;
    memset(a, 0, sz * sizeof(obj*));
  }
  void add(obj* OBJ) {
    if(index >= sz) return;
    a[index++] = OBJ;
  }
  friend class sp;
};

// Iterator:
class sp {
  obj_container* oc;
  int index;
public:
  sp(obj_container* OC) {
    index = 0;
    oc = OC;
  }
  // Return value indicates end of list:
  int operator++() { // Prefix
    if(index >= oc->sz) return 0;
    if(oc->a[++index] == 0) return 0;
    return 1;
  }
  int operator++(int) { // Postfix
    return operator++(); // Use prefix version
  }
  obj* operator->() const {
    if(oc->a[index]) return oc->a[index];
    static obj dummy;
    return &dummy;
  }
};

main() {
  const sz = 10;
  obj o[sz];
  obj_container OC;
  for(int i = 0; i < sz; i++)
    OC.add(&o[i]); // Fill it up
```

```
  sp SP(&OC); // Create an iterator
  do {
    SP->f(); // Smart pointer calls
    SP->g();
  } while(SP++);
}
```

类obj定义了程序中使用的对象。函数f( )和g( )用静态数据成员打印令人感兴趣的值。使用obj_container的函数add( )将指向这些对象的指针储存在类型 obj_container的包容器中。obj_container看起来像一个指针数组，但却发现没有办法得到这些指针。然而，类 sp声明为friend类，所以它允许进入这个包容器内。类sp看起来像一个聪明的指针 —— 可以使用运算符++向前移动它（也可以定义一个运算符 --），它不会超出包容器的范围，它可以返回它指向的内容（通过这个灵巧指针）。注意，循环子(iterator)是与包容器配套使用的 —— 不像指针，没有“ 通用目的 ”的循环子。包容器和循环子将在第 15章深入讨论。

在main( )中，一旦包容器OC装入obj对象，一个循环子SP就创建了。灵巧指针按下面的表达式调用：

```
SP->f( ) ; //Smart pointer calls
SP->g( ) ;
```

这里，尽管SP实际上并不含成员函数f( ) 和g( )，但结构指针机制通过obj*调用这些函数，obj* 是通过sp::operator->返回的。编译器进行所有检查以确信函数调用正确。

虽然，灵巧指针的执行机制比其他运算符复杂一些，但目的是一样的 ---为类的用户提供更为方便的语法。

11.3.5 不能重载的运算符

在可用的运算符集合里存在一些不能重载的运算符。这样限制的通常原因是出于对安全的考虑：如果这些运算符也可以被重载的话，将会造成危害或破坏安全机制，使得事情变得困难或混淆现有的习惯。

现在，成员选择运算符‘ . ’在类中对任何成员都有一定的意义。但如果允许它重载，就不能用普通的方法访问成员，只能用指针和指针运算符 ->访问。

成员指针逆向引用的运算符‘ .* ’因为与运算符‘ . ’同样的原因而不能重载。

没有求幂运算符。大多数通常的选择是从 Fotran语言引用运算符**，但这出现了难以分析的问题。C也没有求幂运算符， C++似乎也不需要，因为这可以通过函数调用来实现。求幂运算符增加了使用的方便，但没有增加新的语言功能，反而给编译器增加了复杂性。

不存在用户定义的运算符，即不能编写目前运算符集合中没有的运算符。不能这样做的部分原因是难以决定其优先级，另一部分原因是没有必要增加麻烦。

不能改变优先级规则。否则人们很难记住它们。

11.4 非成员运算符

在前面的一些例子里，运算符可能是成员运算符或非成员运算符，这似乎没有多大差异。这就会出现一个问题：“ 我应该选择哪一种？ ”总的来说，如果没有什么差异，它们应该是成员运算符。这样做强调了运算符和类的联合。当左侧操作数是当前类的对象时，运算符会工作得很好。

但也不完全是这种情况——有时我们左侧运算符是别的类对象。这种情况通常出现在为iostreams重载运算符<< 和 >>时候。

```
//: IOSOP.CPP -- Iostream operator overloading
// Example of non-member overloaded operators
#include <iostream.h>
#include <strstrea.h>
#include <string.h>
#include "..\allege.h"

class intarray {
  enum { sz = 5 };
  int i[sz];
public:
  intarray() {
    memset(i, 0, sz* sizeof(*i));
  }
  int& operator[](int x) {
    allege(x >= 0 && x < sz,
           "operator[] out of range");
    return i[x];
  }
  friend ostream&
    operator<<(ostream& os,
               const intarray& ia);
  friend istream&
    operator>>(istream& is, intarray& ia);
};

ostream& operator<<(ostream& os,
                    const intarray& ia){
  for(int j = 0; j < ia.sz; j++) {
    os << ia.i[j];
    if(j != ia.sz -1)
      os << ", ";
  }
  os << endl;
  return os;
}

istream& operator>>(istream& is, intarray& ia){
  for(int j = 0; j < ia.sz; j++)
    is >> ia.i[j];
  return is;
}

main() {
```

```
  istrstream input("47 34 56 92 103");
  intarray I;
  input >> I;
  I[4] = -1; // Use overloaded operator[]
  cout << I;
}
```

这个类也包含重载运算符[]，这个运算符在数组里返回了一个合法值的引用。一个引用被返回，所以下面的表达式：

```
I[4] = -1 ;
```

看起来不仅比使用指针更规范些，而且它也达到了预期的效果。

被重载的移位运算符通过引用方式传递和返回，所以运算将影响外部对象。在函数定义中，表达式像

```
os << ia.i[j] ;
```

会使现有的重载运算符函数被调用（即那些定义在 IOSTREAM.H中的）。在这个情况下，被调用的函数是ostream& operator<<(ostream&,int)，这是因为ia.i[j]是一个int值。

一旦所有的动作在istream或ostream上完成，它将被返回，因此它可被用于更复杂的表达式。

这个例子使用的是插入符和提取符的标准形式。如果我们想为自己的类创建一个集合，可以拷贝这个函数署名和返回类型，并遵从它的体的形式。

基本方针

Murry [1] 为在成员和非成员之间的选择提出了如下的方针：

运算符	建议使用
所有的一元运算符	成员
= ( ) [] ->	必须是成员
+= -= /= *= ^= &= |= %= >>= <<=	成员
所有其他二元运算符	非成员

11.5 重载赋值符

赋值符在C++中常常产生混淆。这是毫无疑问的，因为‘ = ’在编程中是最基本的运算符，是在机器层上拷贝寄存器。另外，当使用‘ = ’时也能引起拷贝构造函数（上一章内容）调用：

```
foo B ;
foo A = B ;
A = B ;
```

第2行定义了对象A。一个新对象先前不存在，现在正被创建。因为我们现在知道了 C++编译器关于对象初始化是如何保护的，所以知道在对象被定义的地方构造函数总是必须被调用。但是哪个构造函数呢？A是从现有的foo对象创建的，所以只有一个选择：拷贝构造函数。所以虽然这里只包括一个‘ = ’，但拷贝构造函数仍被调用。

[1] Rob Murray, C++ Strategies & Tactics, Addision-Wesley, 1993, page 47.

第3行情况不同了。在‘ = ’左侧有一个以前初始化了的对象。很清楚，不用为一个已经存在的对象调用构造函数。在这种情况下，为 A调用foo::operator=，把foo::operator=右侧的任何东西作为参数。（我们可以有多种取不同右侧参数的operator=函数）

对于拷贝构造函数没有这个限制。我们在任何时候使用一个‘ = ’代替普通形式的构造函数调用来初始化一个对象时，无论=右侧是什么，编译器都会为我们寻找一个构造函数：

```
//: FEEFI.CPP -- Copying vs. initialization

class fi {
public:
  fi() {}
};

class fee {
public:
  fee(int) {}
  fee(const fi&) {}
};

main() {
  fee f = 1; // fee(int)
  fi FI;
  fee fum = FI; // fee(fi)
}
```

当处理‘ = ’时，记住这个差别是非常重要的：如果对象还没有被创建，初始化是需要的，否则使用赋值运算符‘ = ’。

对于初始化，使用‘ = ’可以避免写代码。但这要用显式的构造函数形式。于是最后一行应写成：

```
fee fum(FI) ;
```

这个方法可以避免使读者混淆。

运算符‘ = ’的行为

在BINARY.CPP中，我们看到运算符‘ = ’仅是成员函数，它密切地与‘ = ’左侧的对象相联系。如果允许我们全局性地定义运算符‘ = ’，那么我们就会试图重新定义内置的‘ = ’：

```
int operator=(int,foo) ; // global = not allowed!
```

这是绝对不允许的，编译器通过强制运算符‘ = ’为成员函数而避开这个问题。

当创建一个运算符‘ = ’时，必须从右侧对象中拷贝所有需要的信息完成为类的“ 赋值 ”，对于单个对象，这是显然的：

```
//: SIMPCOPY.CPP -- Simple operator=()
#include <iostream.h>

class value {
  int a, b;
```

```
  float c;
public:
  value(int A = 0, int B = 0, float C = 0.0) {
    a = A;
    b = B;
    c = C;
  }
  value& operator=(const value& rv) {
    a = rv.a;
    b = rv.b;
    c = rv.c;
    return *this;
  }
  friend ostream&
    operator<<(ostream& os, const value& rv) {
      return os << "a = " << rv.a << ", b = "
        << rv.b << ", c = " << rv.c;
    }
};

main() {
  value A, B(1, 2, 3.3);
  cout << "A: " << A << endl;
  cout << "B: " << B << endl;
  A = B;
  cout << "A after assignment: " << A << endl;
}
```

这里，‘ = ’左侧的对象拷贝了右侧对象中的所有内容，然后返回它的引用，所以我们还可以创建更加复杂的表达式。

这个例子犯了一个普通的错误。当准备给两个相同类型的对象赋值时，应该首先检查一下自赋值(self-assignment)：这个对象是否对自身赋值了？在一些情况下，例如本例，无论如何执行这些赋值运算都是无害的，但如果对类的实现作了修改，那么将会出现差异。如果我们习惯于不做检查，就可能忘记并产生难以发现的错误。

1. 类中指针

如果对象不是如此简单时将会发生什么事情？例如，如果对象里包含指向别的对象的指针将如何？简单地拷贝一个指针意味着以指向相同的存储单元的对象而结束。这种情况，就需要自己做注释记住这点。

这里有两个解决问题的方法。当我们做一个赋值运算或一个拷贝构造函数时，最简单的技术是拷贝这个指针所涉及的一切，这是非常简单的。

```
//: COPYMEM.CPP -- Duplicate during assignment
#include <stdlib.h>
#include <string.h>
#include "..\allege.h"
```

```
class withPointer {
  char* p;
  enum { blocksz = 100 };
public:
  withPointer() {
    p = (char*)malloc(blocksz);
    allegemem(p);
    memset(p, 1, blocksz);
  }
  withPointer(const withPointer& wp) {
    p = (char*)malloc(blocksz);
    allegemem(p);
    memcpy(p, wp.p, blocksz);
  }
  withPointer&
  operator=(const withPointer& wp) {
    // Check for self-assignment:
    if(&wp != this)
      memcpy(p, wp.p, blocksz);
    return *this;
  }
  ~withPointer() {
    free(p);
  }
};

main() {}
```

这里展示了当我们的类包含了指针时，总是需要定义的 4个函数：所有必需的普通构造函数、拷贝构造函数、运算符‘ =’(无论定义它还是不允许它）和析构函数。对运算符‘ =’当然要检查自赋值，虽然这儿不需要，但我们应养成这种习惯。这实际上减少了改变代码而忘记检查自赋值的可能性。

这里，构造函数分配存储单元并对它初始化，运算符‘ =’拷贝它，析构函数释放存储单元。然而，如果要处理许多存储单元并对其初始化时，我们也许想避免这种拷贝。解决这个问题的通常方法被称为引用记数 (reference counting)。可以使一块存储单元具有智能，它知道有多少对象指向它。拷贝构造函数或赋值运算意味着把另外的指针指向现在的存储单元并增加引用记数。消除意味着减小引用记数，如果引用记数为 0意味着销毁这个对象。

但如果向这块存储单元写入将会如何呢？因为不止一个对象使用这块存储单元，所以当我们修改自己的存储单元时，也等于也修改了他人的存储单元。为了解决这个问题，经常使用另外一个称为写拷贝(copy-on-write)的技术。在向这块存储单元写之前，应该确信没有其他人使用它。如果引用记数大于 1，在写之前必须拷贝这块存储单元，这样就不会影响他人了。这儿提供了一个简单的引用记数和关于写拷贝的例子：

```
//: REFCOUNT.CPP -- Reference count, copy-on-write
#include <string.h>
```

```
#include <assert.h>

class counted {
  class memblock {
    enum { size = 100 };
    char c[size];
    int refcount;
  public:
    memblock() {
      memset(c, 1, size);
      refcount = 1;
    }
    memblock(const memblock& rv) {
      memcpy(c, rv.c, size);
      refcount = 1;
    }
    void attach() { ++refcount; }
    void detach() {
      assert(refcount != 0);
      // Destroy object if no one is using it:
      if(--refcount == 0) delete this;
    }
    int count() const { return refcount; }
    void set(char x) { memset(c, x, size); }
    // Conditionally copy this memblock.
    // Call before modifying the block; assign
    // resulting pointer to your block;
    memblock* unalias() {
      // Don't duplicate if not aliased:
      if(refcount == 1) return this;
      --refcount;
      // Use copy-constructor to duplicate:
      return new memblock(*this);
    }
  } * block;
public:
  counted() {
    block = new memblock; // Sneak preview
  }
  counted(const counted& rv) {
    block = rv.block; // Pointer assignment
    block->attach();
  }
  void unalias() { block = block->unalias(); }
  counted& operator=(const counted& rv) {
```

```
    // Check for self-assignment:
    if(&rv == this) return *this;
    // Clean up what you're using first:
    block->detach();
    block = rv.block; // Like copy-constructor
    block->attach();
    return *this;
  }
  // Decrement refcount, conditionally destroy
  ~counted() { block->detach(); }
  // Copy-on-write:
  void write(char value) {
    // Do this before any write operation:
    unalias();
    // It's safe to write now.
    block->set(value);
  }
};

main() {
  counted A, B;
  counted C(A);
  B = A;
  C = C;
  C.write('x');
}
```

嵌套类memblock是被指向的一块存储单元。（注意指针block定义在嵌套类的最后）它包含了一个引用记数及控制和读引用记数的函数。同时这里存在一个拷贝构造函数，所以我们可以从现有的类创建一个新的memblock。

函数attach( )增加一个memblock引用记数用以指示有另一个对象使用它。函数 detach( )减少引用记数。如果引用记数为0，则说明没有对象使用它，所以通过表达式delete this 成员函数销毁它自己的对象。

可以用函数 set( ) 修改存储单元。但在做修改之前，应该确信不是在别的对象使用的memblock上进行。可以通过调用connted::unalias( )，connted::unalias( )调用memblock::unalias( )来做到这点。如果引用记数为 1（意味着没有别的对象指向这块存储单元），后面这个函数将返回block指针，但如果引用记数大于1就要复制这个存储单元。

这个例子已涉及到下一章的内容。C++运算符new和delete代替C语言的malloc( )和free( )来创建和销毁对象。对于这个例子，除了new在分配了存储单元后调用构造函数，delete在释放存储单元之前调用析构函数之外，可以认为new 和delete与malloc( )和free( )一样。

拷贝构造函数给源对象block赋值block，而不是创建它自己的存储单元。然后因为现在增加了使用这个存储单元的对象，所以通过调用 memblock::attach( )增加引用记数。

运算符‘ = ’处理‘ = ’左侧已创建的对象，所以它必须通过为 memblock调用detach( )而首先整理这个存储单元。如果没有其他对象使用它，这个老的 memblock将被销毁。然后运算符

‘=’重复拷贝构造函数的行为。注意它首先检查是否给它本身赋予相同的对象。

析构函数调用detach( )有条件地销毁memblock。

为了实现写拷贝，必须控制所有写存储单元的动作。这意味着不能向外部传递原有指针。我们会说：“告诉我您想做什么，我将为您做！”例如成员函数 write( )允许对这个存储单元修改数值。但它首先必须使用unalias( )防止修改一个已别名化了的存储单元（超过一个对象使用的存储单元）。

在main( )中测试了几个必须正确实现引用记数的函数：构造函数、拷贝构造函数、运算符‘=’和析构函数。在main( )中也通过为对象C调用write( )测试了写拷贝，对象C是已别名化了的A存储单元。

2. 跟踪输出

为了验证这个方案是正确的，最好的方法是对类增加信息和功能以便产生可被分析的跟踪输出。这儿的REFCOUNT.CPP增加了跟踪信息。

```
//: RCTRACE.CPP -- REFCOUNT.CPP w/ trace info
#include <string.h>
#include <fstream.h>
#include <assert.h>
ofstream out("rctrace.out");

class counted {
  class memblock {
    enum { size = 100 };
    char c[size];
    int refcount;
    static int blockcount;
    int blocknum;
  public:
    memblock() {
      memset(c, 1, size);
      refcount = 1;
      blocknum = blockcount++;
    }
    memblock(const memblock& rv) {
      memcpy(c, rv.c, size);
      refcount = 1;
      blocknum = blockcount++;
      print("copied block");
      out << endl;
      rv.print("from block");
    }
    ~memblock() {
    out << "\tdestroying block "
        << blocknum << endl;
  }
  void print(const char* msg = "") const {
```

```
      if(*msg) out << msg << ", ";
      out << "blocknum:" << blocknum;
      out << ", refcount:" << refcount;
    }
    void attach() { ++refcount; }
    void detach() {
      assert(refcount != 0);
      // Destroy object if no one is using it:
      if(--refcount == 0) delete this;
    }
    int count() const { return refcount; }
    void set(char x) { memset(c, x, size); }
    // Conditionally copy this memblock.
    // Call before modifying the block; assign
    // resulting pointer to your block;
    memblock* unalias() {
      // Don't duplicate if not aliased:
      if(refcount == 1) return this;
      --refcount;
      // Use copy-constructor to duplicate:
      return new memblock(*this);
    }
  } * block;
  enum { sz = 30 };
  char id[sz];
public:
  counted(const char* ID = "tmp") {
    block = new memblock; // Sneak preview
    strncpy(id, ID, sz);
  }
  counted(const counted& rv) {
    block = rv.block; // Pointer assignment
    block->attach();
    strncpy(id, rv.id, sz);
    strncat(id, " copy", sz - strlen(id));
  }
  void unalias() { block = block->unalias(); }
  void addname(const char* nm) {
    strncat(id, nm, sz - strlen(id));
  }
  counted& operator=(const counted& rv) {
    print("inside operator=\n\t");
    if(&rv == this) {
      out << "self-assignment" << endl;
```

```
    return *this;
  }
  // Clean up what you're using first:
  block->detach();
  block = rv.block; // Like copy-constructor
  block->attach();
  return *this;
}
// Decrement refcount, conditionally destroy
~counted() {
  out << "preparing to destroy: " << id
      << endl << "\tdecrementing refcount ";
  block->print();
  out << endl;
  block->detach();
}
// Copy-on-write:
void write(char value) {
  unalias();
  block->set(value);
}
void print(const char* msg = "") {
  if(*msg) out << msg << " ";
  out << "object " << id << ": ";
  block->print();
  out << endl;
  }
};

int counted::memblock::blockcount = 0;

main() {
  counted A("A"), B("B");
  counted C(A);
  C.addname(" (C) ");
  A.print();
  B.print();
  C.print();
  B = A;
  A.print("after assignment\n\t");
  B.print();
  out << "Assigning C = C" << endl;
  C = C;
  C.print("calling C.write('x')\n\t");
  C.write('x');
  out << endl << "exiting main()" << endl;
}
```

现在memblock含有一个static数据成员blockcount来记录创建的存储单元号码，为了区分这些存储单元它还为每个存储单元创建了唯一号码（存放在 blocknum中）。在析构函数中声明哪一个存储单元被销毁，print( )函数显示块号和引用记数。

类counted含有一个缓冲器 id用来记录对象信息。 counted构造函数创建了一个新的memblock对象并把结果赋给了block（这个结果是一个堆上指向memblock的指针）。从参数拷贝来的标识符加了一个单词“ copy ”用以显示它是从哪里拷贝来的。函数addname( )也让我们在id（这是实际的标识符，所以我们可以看到它是什么以及从哪里拷贝来的）中加入有关对象的附加信息。

这里是输出结果：

```
object A: blocknum:0, refcount:2
object B: blocknum:1, refcount:1
object A copy (C) : blocknum:0, refcount:2
inside operator=
        object B: blocknum:1, refcount:1
      destroying block 1
after assignment
        object A: blocknum:0, refcount:3
object B: blocknum:0, refcount:3
Assigning C = C
inside operator=
        object A copy (C) : blocknum:0, refcount:3
self-assignment
calling C.write('x')
        object A copy (C) : blocknum:0, refcount:3
copied block, blocknum:2, refcount:1
from block, blocknum:0, refcount:2
exiting main()
preparing to destroy: A copy (C)
      decrementing refcount blocknum:2, refcount:1
      destroying block 2
preparing to destroy: B
      decrementing refcount blocknum:0, refcount:2
preparing to destroy: A
      decrementing refcount blocknum:0, refcount:1
      destroying block 0
```

通过研究输出结果、跟踪源代码和对程序测试，我们会加深对这些技术的理解。

3. 自动创建运算符‘ = ’

因为将一个对象赋给另一个相同类型的对象是大多数人可能做的事情，所以如果没有创建type::operator=(type)，编译器会自动创建一个。这个运算符行为模仿自动创建的拷贝构造函数的行为：如果类包含对象（或是从别的类继承的），对于这些对象，运算符‘ = ’被递归调用。这被称为成员赋值(memberwise assignment)。见如下例子：

```
//: AUTOEQ.CPP -- Automatic operator=()
#include <iostream.h>
```

```
class bar {
public:
  bar& operator=(const bar&) {
    cout << "inside bar::operator=()" << endl;
    return *this;
  }
};

class foo {
  bar B;
};

main() {
  foo a, b;
  a = b; // Prints: "inside bar::operator=()"
}
```

为foo自动生成的运算符‘ = ’调用bar::operator=。

一般我们不会想让编译器做这些。对于复杂的类（尤其是它们包含指针的情况），我们应该显式地创建一个运算符‘ = ’。如果真的不想让人执行赋值运算，可以把运算符‘ = ’声明为private函数。（除非在类内使用它，否则不必定义它。）

11.6 自动类型转换

在C和C++中，如果编译器看到一个表达式或函数调用使用了一个不合适的类型，它经常会执行一个自动类型转换。在C++中，可以通过定义自动类型转换函数来为用户定义类型达到相同效果。这些函数有两种类型：特殊类型的构造函数和重载的运算符。

11.6.1 构造函数转换

如果我们定义一个构造函数，这个构造函数能把另一类型对象（或引用）作为它的单个参数，那么这个构造函数允许编译器执行自动类型转换。如下例：

```
//: AUTOCNST.CPP -- Type conversion constructor

class one {
public:
  one() {}
};

class two {
public:
  two(const one&) {}
};

void f(two) {}
```

```
main() {
  one One;
  f(One); // Wants a two, has a one
}
```

当编译器看到f( )以为对象one参数调用时，编译器检查f( )的声明并注意到它需要一个two对象作为参数。然后，编译器检查是否有从对象 one 到two的方法。它发现了构造函数two::two(one)，two::two(one)被悄悄地调用，结果对象two被传递给f( )。

在这个情况里，自动类型转换避免了定义两个 f( )重载版本的麻烦。然而，代价是隐藏了构造函数对two的调用，如果我们关心f( )的调用效率的话，那就不要使用这种方法。

• 阻止构造函数转换

有时通过构造函数自动转换类型可能出现问题。为了避开这个麻烦，可以通过在前面加关键字explicit [1]（只能用于构造函数）来修改构造函数。上例类two的构造函数作了修改，如下：

```
class one {
public:
  one() {}
};

class two {
public:
  explicit two(const one&) {}
};

void f(two) {}

main() {
  one One;
//!  f(One); // no auto conversion allowed
  f(two(One)); // OK -- user performs conversion
}
```

通过使类two的构造函数显式化，编译器被告知不能使用那个构造函数（那个类中其他非显式化的构造函数仍可以执行自动类型转换）执行任何自动转换。如果用户想进行转换必须写出代码。上面代码f(two(One))创建一个从类型One到two的临时对象，就像编译器在前面版本中做的那样。

11.6.2 运算符转换

第二种自动类型转换的方法是通过运算符重载。我们可以创建一个成员函数，这个函数通过在关键字operator后跟随想要转换到的类型的方法，将当前类型转换为希望的类型。这种形式的运算符重载是独特的，因为没有指定一个返回类型——返回类型就是我们正在重载的运算符的名字。这儿有一个例子：

[1] 写作本书时，explicit是语言中新的关键字。我们的编译器可能还不支持它。

```
//: OPCONV.CPP -- Op overloading conversion

class three {
  int i;
public:
  three(int I = 0, int = 0) : i(I) {}
};

class four {
  int x;
public:
  four(int X) : x(X) {}
  operator three() const { return three(x); }
};

void g(three) {}
main() {
  four Four(1);
  g(Four);
  g(1);  // Calls three(1,0)
}
```

用构造函数技术，目的类执行转换。然而使用运算符技术，是源类执行转换。构造函数技术的价值是在创建一个新类时为现有系统增加了新的转换途径。然而，创建一个单一参数的构造函数总是定义一个自动类型转换（即使它有不止一个参数也是一样，因为其余的参数将被缺省处理），这可能并不是我们所想要的。另外，使用构造函数技术没有办法实现从用户定义类型向内置类型转换，这只有运算符重载可能做到。

• 反身性

使用全局重载运算符而不用成员运算符的最便利的原因之一是在全局版本中的自动类型转换可以针对左右任一操作数，而成员版本必须保证左侧操作数已处于正确的形式。如果想两个操作数都被转换，全局版本可以节省很多代码。这儿有一个小例子。

```
//: REFLEX.CPP -- Reflexivity in overloading

class number {
  int i;
public:
  number(int I = 0) { i = I; }
  const number
  operator+(const number& n) const {
    return number(i + n.i);
  }
  friend const number
    operator-(const number&, const number&);
};
```

```
const number
  operator-(const number& n1,
            const number& n2) {
    return number(n1.i - n2.i);
}

main() {
  number a(47), b(11);
  a + b; // OK
  a + 1; // 2nd arg converted to number
//! 1 + a; // Wrong! 1st arg not of type number
  a - b; // OK
  a - 1; // 2nd arg converted to number
  1 - a; // 1st arg converted to number
}
```

类number有一个成员运算符+号和一个友元(friend)运算符 - 号。因为有一个使用单一int参数的构造函数，int自动转换为number，但这要在正确的条件下。在 main( )里，可以看到增加一个number到另一个number进行得很好，这是因为它重载的运算符非常匹配。当编译器看到一个number后跟一个+号和一个int时，它也能和成员函数number::operator+相匹配并且构造函数把int参数转换为number。但当编译器看到一个int、一个+号和一个number时，它就不知道如何去做，因为它所拥有的是number::operator+，需要左侧的操作数是number对象。因此，编译器发出一个出错信息。

对于友元运算符-号，情况就不同了。编译器需要填满两个参数，它不限定 number作为左侧参数。因此，如果看到表达式 1-a，编译器就使用构造函数把第一个参数转换为 number。有时我们也许想通过把它们设成成员函数来限定运算符的使用。例如当用一个矢量与矩阵相乘，矢量必须在右侧。但如果想让运算符转换任一个参数，就要使运算符为友元函数。

幸运的是编译器不会把表达式 1-1的两个参数转换为number对象，然后调用运算符 - 号。那将意味着现有的C代码可能突然执行不同的工作了。编译器首先匹配“最简单的”可能性，对于表达式1-1将优先使用内部运算符。

11.6.3 一个理想的例子：strings

这是一个自动类型转换对于 string类非常有帮助的例子。如果不用自动类型转换就想从标准的C库函数中使用所有的字符串函数，那么就得为每一个函数写一个相应的成员函数，就像下面的例子：

```
//: STRINGS1.CPP -- No auto type conversion
#include <string.h>
#include <stdlib.h>
#include "..\allege.h"

class string {
  char* s;
public:
  string(const char* S = "") {
```

```
    s = (char*)malloc(strlen(S) + 1);
    allegemem(s);
    strcpy(s, S);
  }
  ~string() { free(s); }
  int Strcmp(const string& S) const {
    return ::strcmp(s, S.s);
  }
  // ... etc., for every function in string.h
  };

  main() {
    string s1("hello"), s2("there");
    s1.Strcmp(s2);
  }
```

这里只写了strcmp( )函数，但必须为STRING.H可能需要的每一个函数写一个相应的函数。幸运的是，可以提供一个允许访问STRING.H中所有函数的自动类型转换：

```
//: STRINGS2.CPP -- With auto type conversion
#include <string.h>
#include <stdlib.h>
#include "..\allege.h"

class string {
  char* s;
public:
  string(const char* S = "") {
    s = (char*)malloc(strlen(S) + 1);
    allegemem(s);
    strcpy(s, S);
  }
  ~string() { free(s); }
  operator const char*() const { return s; }
};

main() {
  string s1("hello"), s2("there");
  strcmp(s1, s2); // Standard C function
  strspn(s1, s2); // Any string function!
}
```

因为编译器知道如何从string转换到char*，所以现在任何一个接受char*参数的函数也可以接受string参数。

11.6.4 自动类型转换的缺陷

因为编译器必须选择如何执行类型转换，所以如果没有正确地设计出转换，编译器会产生

麻烦。类X可以用operator Y( ) 将它本身转换到类Y，这是一个简单且明显的情况。如果类Y有一个单个参数为X的构造函数，也表示同样的类型转换。现在编译器有两个从 X到Y的转换方法，所以发生转换时，编译器会产生一个不明确指示的出错信息：

```
//: AMBIG.CPP -- Ambiguity in type conversion

class Y; // Class declaration

class X {
public:
  operator Y() const; // Convert X to Y
};

class Y {
public:
  Y(X); // Convert X to Y
};

void f(Y);

main() {
  X x;
//! f(x); // Error: ambiguous conversion
}
```

这个问题解决方案不是仅提供一个从一个类型到另一个类型的自动转换路径。

提供自动转换到不止一种类型时，会引发更困难的问题。有时，这个问题被称为扇出(fan-out)：

```
//: FANOUT.CPP -- Type conversion fanout

class A {};
class B {};

class C {
public:
  operator A() const;
  operator B() const;
};

// Overloaded h():
void h(A);
void h(B);

main() {
  C c;
//! h(c); // Error: C -> A or C -> B ???
}
```

类C有向A和B的自动转换。这样存在一个隐藏的缺陷：使用创建的两种版本的重载运算符h( )时问题就出现了。（只有一个版本时，main( )里的代码会正常运行。）

通常，对于自动类型的解决方案是只提供一个从一个类型向另一个类型转换的自动转换版本。当然我们也可以有多个向其他类型的转换，但它们不应该是自动转换，而应该创建显式的调用函数，例如用名字make_A( )和make_B( )表示这些函数。

• 隐藏的行为

自动类型转换会引入比所希望的更多的潜在行为。下面看 11.5节FEEFI.CPP的修改后的例子：

```
//: FEEFI2.CPP -- Copying vs. initialization

class fi {};

class fee {
public:
  fee(int) {}
  fee(const fi&) {}
};

class fo {
  int i;
public:
  fo(int x = 0) { i = x; }
  operator fee() const { return fee(i); }
};

main() {
  fo FO;
  fee fiddle = FO;
}
```

这里没有从fo对象创建fee fiddle的构造函数。然而，fo有一个到fee的自动类型转换。这里也没有从fee对象创建fee的拷贝构造函数，但这是一种能由编译器帮助我们创建的特殊函数之一。（缺省的构造函数、拷贝构造函数、运算符‘ = ’和析构函数可被自动创建）对于下面的声明，自动类型转换运算符被调用并创建一个拷贝函数：

```
fee fiddle = FO ;
```

自动类型转换应该小心使用。它在减少代码方面是非常出色的，但不值得无缘无故地使用。

11.7 小结

运算符重载存在的原因是为了使编程容易。运算符重载没有那么神秘，它只不过是拥有有趣名字的函数。当它以正确的形式出现时，编译器调用这个函数。但如果运算符重载对于类的设计者或类的使用者不能提供特别显著的益处，则最好不要使用，因为增加运算符重载会使问题混淆。

11.8 练习

1. 写一个有重载运算符++的类。试着用前缀和后缀两种形式调用此运算符，看看编译器会给我们什么警告。

2. 写一个只含有单个private char成员的类。重载iostream运算符<<和>>（像在IOSOP.CPP中的一样）并测试它们，可以用fstreams、strstreams和stdiostreams(cin和cout)测试它们。

3. 写一个包含重载的运算符+、-、*、/和赋值符的number类。出于效率考虑，为这些函数合理地选择返回值以便以链式写表达式。写一个自动类型转换运算符 int( )。

4. 合并在UNARY.CPP和BINARY.CPP中的类。

5. 对FANOUT.CPP作如下修改：创建一个显式函数，用它代替自动转换运算符来完成类型转换。

第12章　动态对象创建

有时我们能知道程序中对象的确切数量、类型和生命期。但情况并不总是这样。

空中交通系统必须处理多少架飞机？一个CAD系统需要多少个形状？在一个网络中有多少个节点？

为了解决这个普通的编程问题，在运行时能创建和销毁对象是基本的要求。当然， C已提供了动态内存分配函数 malloc( )和free( )（以及malloc( )的变种），这些函数在运行时从堆中（也称自由内存）分配存储单元。

然而，在C++中这些函数不能很好地运行。构造函数不允许通过向对象传递内存地址来初始化它。如果那么做了，我们可能

1) 忘记了。则对象初始化在C++中难以保证。

2) 期望某事发生，但结果是在给对象初始化之前意外地对对象作了某种改变。

3) 把错误规模的对象传递给了它。

当然，即使我们把每件事都做得很正确，修改我们的程序的人也容易犯同样的错误。不正确的初始化是编程出错的主要原因，所以在堆上创建对象时，确保构造函数调用是特别重要的。

C++是如何保证正确的初始化和清理并允许我们在堆上动态创建对象的呢？

答案是使动态对象创建成为语言的核心。 malloc( )和free( )是库函数，因此不在编译器控制范围之内。如果我们有一个能完成动态内存分配及初始化工作的运算符和另一个能完成清理及释放内存工作的运算符，编译器就可以保证所有对象的构造函数和析构函数都会被调用。

在本章中，我们将明白C++的new和delete是如何通过在堆上安全创建对象来出色地解决这个问题的。

12.1　对象创建

当一个C++对象被创建时，有两件事会发生。

1) 为对象分配内存。

2) 调用构造函数来初始化那个内存。

到目前为止，我们应该确保步骤 2）一定发生。C++强迫这样做是因为未初始化的对象是程序出错的主要原因。不用关心对象在哪里创建和如何创建的 —— 构造函数总是被调用。

然而，步骤1）可以以几种方式或在可选择的时间内发生：

1) 静态存储区域，存储空间在程序开始之前就可以分配。这个存储空间在程序的整个运行期间都存在。

2) 无论何时到达一个特殊的执行点（左花括号）时，存储单元都可以在栈上被创建。出了执行点（右花括号），这个存储单元自动被释放。这些栈分配运算内置在处理器的指令集中，非常有效。然而，在写程序的时候，必须知道需要多少个存储单元，以使编译器生成正确的指令。

3) 存储单元也可以从一块称为堆（也可称为自由存储单元）的地方分配。这称为动态内存分配，在运行时调用程序分配这些内存。这意味着可以在任何时候分配内存和决定需要多少内

存。我们也负责决定何时释放内存。这块内存的生存期由我们选择决定——而不受范围限制。

这三个区域经常被放在一块连续的物理存储单元里：静态内存、堆栈和堆（由编译器的作者决定它们的顺序），但没有一定的规则。堆栈可以在某一特定的地方，堆的实现可以通过调用由运算系统分配的一块存储单元来完成。对于一个程序设计者，这三件事无须我们来完成，所以我们所要思考的是什么时候申请内存。

12.1.1 C从堆中获取存储单元的方法

为了在运行时动态分配内存，C在它的标准库函数中提供了一些函数：从堆中申请内存的函数malloc( )以及它的变种calloc( )和realloc( )；释放内存返回给堆的函数free( )。这些函数是有效的但较原始，需要编程人员理解和小心使用。对使用C的动态内存分配函数创建一个类的实例，必须做：

```
//: MALCLASS.CPP -- Malloc with class objects
// What you'd have to do if not for "new"
#include <stdlib.h> // Malloc() & free()
#include <string.h> // Memset()
#include "..\allege.h"
#include <iostream.h>

class obj {
  int i, j, k;
  enum { sz = 100 };
  char buf[sz];
public:
  void initialize() { // Can't use constructor
    cout << "initializing obj" << endl;
    i = j = k = 0;
    memset(buf, 0, sz);
  }
  void destroy() { // Can't use destructor
    cout << "destroying obj" << endl;
  }
};

main() {
  obj* Obj = (obj*)malloc(sizeof(obj));
  allegemem(Obj);
  Obj->initialize();
  // ... sometime later:
  Obj->destroy();
  free(Obj);
}
```

在下面这行代码中，我们可以看到使用malloc( )为对象分配内存：

```
obj* Obj = (obj*)malloc(sizeof(obj)) ;
```

这里用户必须决定对象的长度（这也是程序出错原因之一）。因为它是一块内存而不是一个对象，所以malloc( )返回一个void*。C++不允许将一个void* 赋予任何指针，所以必须映射。

因为malloc( )可能找不到可分配的内存（在这种情况下它返回 0），所以必须检查返回的指针以确信内存分配成功。

但最坏的是：

```
Obj->initialize( ) ;
```

用户在使用对象之前必须记住对它初始化。注意构造函数没有被使用，因为构造函数不能被显式地调用 —— 而是当对象创建时由编译器调用。这里的问题是现在用户可能在使用对象时忘记执行初始化，因此这也是引入程序缺陷的主要来源。

许多程序设计者发现C的动态内存分配函数太复杂，令人混淆。所以， C程序设计者常常在静态内存区域使用虚拟内存机制分配很大的变量数组以避免使用动态内存分配。因为 C++能让一般的程序员安全使用库函数而不费力，所以应当避免使用 C的动态内存方法。

12.1.2 运算符new

C++中的解决方案是把创建一个对象所需的所有动作都结合在一个称为 new的运算符里。当用new（new的表达式）创建一个对象时，它就在堆里为对象分配内存并为这块内存调用构造函数。因此，如果我们写出下面的表达式

```
foo *fp = new foo(1,2) ;
```

在运行时等价于调用 malloc(sizeof(foo))，并使用（1，2）作为参数表来为 foo调用构造函数，返回值作为this指针的结果地址。在该指针被赋给 fp之前，它是不定的、未初始化的对象 —— 在这之前我们甚至不能触及它。它自动地被赋予正确的 foo类型，所以不必进行映射。

缺省的new还检查以确信在传递地址给构造函数之前内存分配是成功的，所以我们不必显式地确定调用是否成功。在本章后面，我们将会发现，如果没有可供分配的内存会发生什么事情。

我们可以为类使用任何可用的构造函数而写一个 new表达式。如果构造函数没有参数，可以写没有构造函数参数表的new表达式：

```
foo *fp = new foo ;
```

我们已经注意到了，在堆里创建对象的过程变得简单了 —— 只是一个简单的表达式,它带有内置的长度计算、类型转换和安全检查。这样在堆里创建一个对象和在栈里创建一个对象一样容易。

12.1.3 运算符delete

new表达式的反面是delete表达式。delete表达式首先调用析构函数，然后释放内存（经常是调用free( )）。正如new表达式返回一个指向对象的指针一样， delete表达式需要一个对象的地址。

```
delete fp ;
```

上面的表达式清除了早先创建的动态分配的对象 foo。

delete只用于删除由new创建的对象。如果用malloc( )（或calloc( )或realloc( )）创建一个对象，然后用delete删除它，这个行为是未定义的。因为大多数缺省的 new和delete实现机制都使

用了malloc( )和free( )，所以我们很可能会没有调用析构函数就释放了内存。

如果正在删除的对象指针是0，将不发生任何事情。为此，建议在删除指针后立即把指针赋值为0以免对它删除两次。对一个对象删除两次一定不是一件好事，这会引起问题。

12.1.4 一个简单的例子

这个例子显示了初始化发生的情况：

```
//: NEWDEL.CPP -- Simple demo of new & delete
#include <iostream.h>

class tree {
  int height;
public:
  tree(int Height) {
    height = Height;
  }
  ~tree() { cout << "*"; }
  friend ostream&
  operator<<(ostream& os, const tree* t) {
    return os << "tree height is: "
              << t->height << endl;
  }
};
main() {
  tree* T = new tree(40);
  cout << T;
  delete T;
}
```

我们通过打印tree的值以证明构造函数被调用了。这里是通过重载运算符 <<和一个ostream一起使用来实现这个运算的。注意，虽然这个函数被声明为一个友元（ friend）函数，但它还是被定义为一个内联函数。这仅仅是为了方便——定义一个友元函数为内联函数不会改变友元状态而且它仍是全局函数而不是一个类的成员函数。同时也要注意返回值是整个输出表达式的结果，它本身是一个ostream&（为了满足函数返回值类型，它必须是ostream&）。

12.1.5 内存管理的开销

当我们在堆里动态创建对象时，对象的大小和它们的生存期被正确地内置在生成的代码里，这是因为编译器知道确切的数量和范围。在堆里创建对象还包括另外的时间和空间的开销。这儿提供了一个典型的方案。（我们可以用calloc( )或realloc( )代替malloc( )）

1) 调用malloc( )，这个函数从堆里申请一块内存。

2) 从堆里搜索一块足够满足请求的内存。可以通过检查显示内存使用情况的某种图或目录来实现搜索。这个过程很快但可能要试探几次，所以它可能是不确定的——即不必指望malloc( )花费完全相同的时间。

3) 在指向这块内存的指针返回之前，这块内存大小和地址必须记录下来，这样以后调用malloc( )时就不会再使用它了，而且当我们调用free( )时，系统就会知道需要释放多大的内存。

实现这些运算的方法可能变化很大。例如，没有办法阻止在处理器里执行原始的内存分配。如果好奇的话，你可以写一个测试程序来猜测 malloc( )实现的方法；也可以读一读库函数的源代码（如果有的话）。

12.2 重新设计前面的例子

现在已经介绍了new和delete（以及其他许多主题）。对于本书前面的stash和stack例子，我们可以使用到目前为止讨论的所有的技术来重写。检查这个新代码将有助于对这些主题的复习。

12.2.1 仅从堆中创建string类

此处，类stash和stack自己都将不“拥有”它们指向的对象。即当 stash或stack出了范围，它也不会为它指向的对象调用 delete。试图使它们成为普通的类是不可能的，原因是它们是void指针。如果delete一个void指针，唯一发生的事是释放了内存，这是因为既没有类型信息也没有办法使得编译器知道要调用哪个析构函数。当一个指针从 stash或stack对象返回时，在使用它之前必须将它做类型映射。这个问题将在13章和15章讨论。

因为包容器自己不拥有指针，所以用户必须对它负责。这意味着在一个包容器上增加一个指向在栈上创建的对象的指针或增加一个指向在同一个包容器堆上创建的对象的指针时将会发生严重的问题。因为delete表达式对于不在堆上分配的指针是不安全的。（从包容器取回一个指针时，如何知道它的对象已经在哪儿分配了内存呢？）为了在如下一个简单的 String类的版本中解决这个问题，下面采取了一些步骤以防止在堆以外的地方创建 String：

```
//: STRINGS.H -- Simple string class
// Can only be built on the heap
#ifndef STRINGS_H_
#define STRINGS_H_
#include <string.h>
#include <iostream.h>

class String {
  char* s;
  String(const char* S) {
    s = new char[strlen(S) + 1];
    strcpy(s, S);
  }
  // Prevent copying:
  String(const String&);
  void operator=(String&);
public:
  // Only make Strings on the heap:
  friend String* makeString(const char* S) {
    return new String(S);
  }
  // Alternate approach:
  static String* make(const char* S) {
```

```
    return new String(S);
  }
  ~String() { delete s; }
  operator char*() const { return s;}
   char* str() const { return s; }
   friend ostream&
     operator<<(ostream& os, const String& S) {
       return os << S.s;
   }
};
#endif // STRINGS_H_
```

为了限制用户使用这个类，主构造函数声明为 private，所以，除了我们以外，没有人可以使用它。另外，拷贝构造函数也声明为 private，但没有定义，以防止任何人使用它，运算符‘ = ’也是如此。用户创建对象的唯一方法是调用一个在堆上创建 String（所以可以知道所有的String都是在堆上创建的）并返回它的指针的特殊函数。

访问这个函数的方法有两种。为了使用简单，它可以是全局的friend函数（称为makeString( )）。但如果不想“ 污染 ”全局名字空间，可以使之为 static成员函数（称为 make( )）并通过String::make( )调用它。后一种形式对于更加明显地表示它属于这个类是有好处的。

在构造函数中，注意表达式：

```
s = new char[strlen(S) +1] ;
```

方括号意味着一个对象数组被创建（此处是一个 char数组），方括号里的数字表示将创建的对象的数量。这就是在程序运行时如何创建一个数组的方法。

对char*类型的自动转换意味着在任何需要 char*的地方都可以使用一个 String对象。另外，一个iostream输出运算符扩充了 iostream库，使得它能够处理String对象。

12.2.2 stash指针

在第5章看到的类stash版本现已被修改，它反映了第 5章以来所介绍的新技术。另外，新的pstash拥有指向在堆中本来就存在的对象的指针，但在第 5章和它前面章节中，旧的stash是拷贝对象到stash包容器里的。有了新介绍的 new和delete，控制指向在堆中创建的对象的指针就变得安全、容易了。

下面提供了“ pointer stash ”的头文件：

```
//: PSTASH.H -- Holds pointers instead of objects
#ifndef PSTASH_H_
#define PSTASH_H_

class pstash {
  int quantity; // Number of storage spaces
  int next; // Next empty space
   // Pointer storage:
  void** storage;
  void inflate(int increase);
public:
```

```
  pstash() {
    quantity = 0;
    storage = 0;
    next = 0;
  }
  // No ownership:
  ~pstash() { delete storage; }
  int add(void* element);
  void* operator[](int index) const; // Fetch
  // Number of elements in stash:
  int count() const { return next; }
};
#endif // PSTASH_H_
```

基本的数据成分是非常相似的，但现在 storage是一个void指针数组，并且用 new代替malloc( )为这个数组分配内存。在下面这个表达式中，对象的类型是void*，所以这个表达式表示分配了一个void指针的数组。

```
storage = new void*[quantity = Quantity] ;
```

析构函数删除void指针本身，而不是试图删除它们所指向的内容（正如前面指出的，释放它们的内存不调用析构函数，这是因为一个void指针没有类型信息）。

其他方面的变化是用运算符[]代替了函数fetch( )，这在语句构成上显得更有意义。因为返回一个void指针，所以用户必须记住在包容器内存储的是什么类型，在取回它们时要映射这些指针（这是在以后章节将要修改的问题）。

下面是成员函数的定义：

```
//: PSTASH.CPP -- Pointer stash definitions
#include "..\11\pstash.h"
#include <iostream.h>
#include <string.h> // Mem functions

int pstash::add(void* element) {
  const InflateSize = 10;
  if(next >= quantity)
    inflate(InflateSize);
  storage[next++] = element;
  return(next - 1); // Index number
}

// Operator overloading replacement for fetch
void* pstash::operator[](int index) const {
  if(index >= next || index < 0)
    return 0;  // Out of bounds
  // Produce pointer to desired element:
  return storage[index];
}
```

```
void pstash::inflate(int increase) {
  const psz = sizeof(void*);
  // realloc() is cleaner than this:
  void** st = new void*[quantity + increase];
  memset(st, 0, (quantity + increase) * psz);
  memcpy(st, storage, quantity * psz);
  quantity += increase;
  delete storage; // Old storage
  storage = st; // Point to new memory
}
```

除了用储存指针代替整个对象的拷贝外，函数 add( )和以前一样有效。拷贝整个对象对于普通对象来说，实际上只需要拷贝构造函数。

实际上，函数 inflate( )代码比先前的版本更复杂且更低效。这是因为先前使用的函数realloc( )可以调整现有内存块的大小，也可以自动地把旧的内存块内容拷贝到更大的内存块里。在任何一件事中我们都不用为它担心，如果不用移动内存，它将进行得更快。因为 new和realloc不等价，所以在这个例子中必须分配一个更大的内存块、执行拷贝和删除旧的内存块。在这种情况下，使用malloc( )、realloc( )和free( )比new和delete在实现方面可能更有意义。幸运的是，这些实现是隐藏的，所以用户可以不用知道这些变化。只要不对同一块内存混合调用，malloc( )家族函数在和new( )及delete( )并行使用时能够保证相互之间的安全。所以这些是完全可以做到的。

• 一个测试程序

下面是为了测试pstash而将stash的测试程序重写后的程序：

```
//: PSTEST.CPP -- Test of pointer stash
#include "..\11\pstash.h"
#include "..\11\strings.h"
#include <fstream.h>
#include "..\allege.h"

main() {
  pstash intStash;
  // new works with built-in types, too:
  for(int i = 0; i < 25; i++)
    intStash.add(new int(i)); // Pseudo-constr.
  for(int u = 0; u < intStash.count(); u++)
    cout << "intStash[" << u << "] = "
         << *(int*)intStash[u] << endl;

  ifstream infile("pstest.cpp");
  allegefile(infile);
  const bufsize = 80;
  char buf[bufsize];
  pstash stringStash;
  // Use global function makeString:
  for(int j = 0; j < 10; j++)
```

```
    if(infile.getline(buf, bufsize))
      stringStash.add(makeString(buf));
  // Use static member make:
  while(infile.getline(buf, bufsize))
    stringStash.add(String::make(buf));
  // Print out the strings:
  for(int v = 0; stringStash[v]; v++) {
    char* p = *(String*)stringStash[v];
    cout << "stringStash[" << v << "] = "
         << p << endl;
  }
}
```

和前面一样，stash被创建并添加了信息。但这次的信息是由 new表达式产生的指针。注意下面这一行：

```
intStash.add( new int(i) ) ;
```

这个表达式new int(i) 使用了伪构造函数形式，所以一个新的 int对象的内存将在堆上创建并且这个int对象被初始化为值i。

注意在打印的时候，由 pstash::operator[]返回的值必须被映射成正确的类型，对于这个程序其他部分的pstash对象，也将重复这个动作。这是使用 void指针作为表达式而出现的不希望的出现效果，将在后面的章节中解决。

第2步测试打开源程序文件并把它读到另一个 psatsh里，把每一行转换为一个 String对象。我们可以看到，makeString( )和String::make( )都被使用以显示两者之间的差别。静态(static)成员函数可能是更好的方法，因为它更明显。

当取回指针时，我们可以看到如下表达式：

```
char* p = *(String*)stringStash[i] ;
```

由运算符[]返回产生的指针必须被映射为 String*以使之具有正确的类型。然后， String*被逆向引用，所以表达式可对一个对象求值。此时，当编译器想要一个 char*时，将看到一个String对象，所以它在String里调用自动类型转换运算符来产生一个char*。

在这个例子里，在堆上创建的对象永远不被销毁。这并没有害处，因为当程序结束时内存会被释放，但在实际情况下我们并不想这样，这个问题将在以后的章节里予以修正。

12.2.3 stack例子

stack例子用了许多自第4章以来介绍的技术。下面是新的头文件。

```
//: STACK11.H -- New version of stack
#ifndef STACK11_H_
#define STACK11_H_

class stack {
  struct link {
    void* data;
    link* next;
    link(void* Data, link* Next) {
```

```
      data = Data;
      next = Next;
    }
  } * head;
public:
  stack() { head = 0; }
  ~stack();
  void push(void* Data) {
    head = new link(Data,head);
  }
  void* peek() const { return head->data; }
  void* pop();
};
#endif // STACK11_H_
```

嵌套的struct link现在可以有自己的构造函数，因为在stack::push( )里， new的使用可以安全地调用那个构造函数。（注意语法很纯正，这将减小了出错的可能。）link::link构造函数简单地初始化了data和next指针，所以在stack::push( )里，下面这行不仅给新的link分配内存，而且巧妙地为link初始化：

```
head = new link(Data, head) ;
```

其余部分的逻辑实际上和第4章是一样的。下面是剩下的两个非内联函数的实现内容。

```
//: STACK11.CPP -- New version of stack
#include <stdlib.h>
#include "..\11\stack11.h"

void* stack::pop() {
  if(head == 0) return 0;
  void* result = head->data;
  link* oldHead = head;
  head = head->next;
  delete oldHead;
  return result;
}

stack::~stack() {
  link* cursor = head;
  while(head) {
    cursor = cursor->next;
    delete head;
    head = cursor;
  }
}
```

唯一的不同是在析构函数里使用delete代替free( )。

跟stash一样，使用void指针意味着在堆上创建的对象不能被stack销毁，所以，如果用户对

stack里的指针不加以管理的话，可能出现不希望的内存漏洞。可以在下面的测试程序里看到这些：

```
//: STKTST11.CPP -- Test new stack
#include "..\11\stack11.h"
#include "..\11\strings.h"
#include <fstream.h>
#include "..\allege.h"

main() {
  // Could also use command-line argument:
  ifstream file("stktst11.cpp");
  allegefile(file);
  const bufsize = 100;
  char buf[bufsize];
  stack textlines;
  // Read file and store lines in the stack:
  while(file.getline(buf,bufsize))
    textlines.push(String::make(buf));
  // Pop lines from the stack and print them:
  String* s;
  while((s = (String*)textlines.pop()) != 0)
    cout << *s << endl;
}
```

与stash例子一样，先打开一个文件，文件的每一行转换成为一个 String对象并存放在stack里，然后打印出来。这个程序没有删除 stack里的指针，stack本身也没有做，所以那块内存丢失了。

12.3 用于数组的new和delete

在C++里，同样容易在栈或堆上创建一个对象数组。当然，应当为数组里的每一个对象调用构造函数。有一个约束：除了在栈上整体初始化（见第 4章）外必须有一个缺省的构造函数，因为不带参数的构造函数必须被每一个对象调用。

当使用new在堆上创建对象数组时，还必须做一些事情。下面有一个对象数组的例子：

```
foo* fp = new foo[100] ;
```

这样在堆上为100个foo对象分配了足够的内存并为每一个对象调用了构造函数。现在，我们拥有一个foo*。它和用如下表达式创建单个对象得到的结果是一样的：

```
foo* fp2 = new foo ;
```

因为这是我们自己写的代码，并且我们知道 fp实际上是一个数组的起始地址，所以用 fp[2]选择数组的元素是有意义的。销毁这个数组时发生了什么呢？下面的语句看起来是完全一样的：

```
delete fp2 ; //OK
delete fp ;  // Not the desired effect
```

并且效果也会一样：为所给地址指向的 foo对象调用析构函数，然后释放内存。对于 fp2，这样是正确的，但对于fp，另外99个析构函数调用没有进行。正确的存储单元数量还会被释放，因

为它被分配在一个大块的内存里，整个内存块的大小被分配程序藏在某处。

解决办法是需要给编译器一个数组起始地址的信息。这可以用下面的语法来实现：

```
delete []fp ;
```

空的方括号告诉编译器产生从数组创建时存放的地方取回数组中对象数量的代码，并为数组的所有对象调用析构函数。这实际上是早期语法的改良形式，偶尔仍可以在老的代码里看到如下的代码：

```
delete [100]fp ;
```

这个语法强迫程序设计者在数组里包含对象的数量，程序设计者有可能把对象数量弄错。而让编译器处理这件事的附加代价是很低的，所以只在一个地方指明对象数量要比在两个地方指明好些。

使指针更像数组

作为题外话，上面定义的fp可以被修改指向任何类型，但这对于一个数组的起始地址来讲没有什么意义。一般讲来，把它定义为常量更有意义些，因为这样任何修改指针的企图都会被指出出错。为了得到这个效果，我们可以试着用下面的表达式：

```
int const* q = new int[10] ;
```

或

```
const int* q = new int[10] ;
```

但在上面这两种情况里，const将和int 捆绑在一起，限定指针指向的内容而不是指针本身。必须用下面的表达式代替：

```
int* const q = new int[10] ;
```

现在在q里的数组元素可以被修改，但对q本身的修改（例如q++）是不合法的，因为它是一个普通数组标识符。

12.4 用完内存

当运算符new找不到足够大的连续内存块来安排对象时将会发生什么？一个称为 new-handler的函数被调用。或者，检查指向函数的指针，如果指针非0，那么它指向的函数被调用。

对于new-handler的缺省动作是抛出一个异常(throw an exception)，这个主题在第17章介绍。然而，如果我们在程序里用堆分配，至少要用“内存已用完”的信息代替new-handler，并异常中断程序。用这个办法，在调试程序时会得到程序出错的线索。对于最终的程序，我们总想使之具有很强的容错性。

通过包含NEW.H，然后以我们想装入的函数地址为参数调用set_new_handler( )函数，这样就替换了new-handler。

```
//: NEWHANDL.CPP -- Changing the new-handler
#include <iostream.h>
#include <stdlib.h>
#include <new.h>

void out_of_memory() {
  cerr << "memory exhausted!" << endl;
  exit(1);
```

```
}

main() {
  set_new_handler(out_of_memory);
  while(1)
    new int[1000]; // Exhausts memory
}
```

new-handler 函数必须不带参数且具有void返回值。while循环将持续分配int对象（并丢掉它们的返回地址）直到全部内存被耗尽。在紧接下去对 new调用时，将没有内存可被分配，所以调用new-handler。

当然，可以写更圆满的new-handler，甚至它可以收回内存（通常叫做垃圾回收）。这不是编程新手的工作。

12.5 重载new和delete

当创建一个new表达式时有两件事发生。首先，使用运算符 new分配内存，然后调用构造函数。在delete表达式里，调用析构函数，然后使用运算符 delete释放内存。我们永远无法控制构造函数和析构函数的调用（否则我们可能意外地搅乱它们），但可以改变内存分配函数运算符new和delete。

被new和delete使用的内存分配系统是为通用目的而设计的。但在特殊的情形下，它不能满足我们的需要。改变分配系统的原因是考虑效率：我们也许要创建和销毁一个特定的类的非常多的对象以至于这个运算变成了速度的瓶颈。 C++允许重载new和delete来实现我们自己的存储分配方案，所以可以像这样处理问题。

另外一个问题是堆碎片：分配不同大小的内存可能造成在堆上产生很多碎片，以至于很快用完内存。也就是内存可能还有，但由于是碎片，找不到足够大的内存满足我们的需要。通过为特定类创建我们自己的内存分配器，可以确保这种情况不会发生。

在嵌入和实时系统里，程序可能必须在有限的资源情况下运行很长时间。这样的系统也可能要求分配内存花费相同的时间且不允许出现堆内存耗尽或出现很多碎片的情况。由客户定制的内存分配器是一种解决办法，否则程序设计者在这种情况下要避免使用 new和delete，从而失去了C++很有价值的优点。

当重载运算符new和delete时，记住只改变原有的内存分配方法是很重要的。编译器将用new代替缺省的版本去分配内存，然后为那个内存调用构造函数。所以，虽然编译器遇到 new时会分配内存并调用构造函数，但当我们重载 new时，可以改变的只是内存分配部分。（delete也有相似的限制。）

当重载运算符new时，也可以替换它用完内存时的行为，所以必须在运算符 new里决定做什么：返回0、写一个调用new-handler的循环、再试着分配或用一个 bad_alloc异常处理（在第17章中讨论）。

重载new和delete与重载任何其他运算符一样。但可以选择重载全局内存分配函数，或为特定的类使用特定的分配函数。

12.5.1 重载全局new和delete

当全局版本的new和delete不能满足整个系统时，对其重载是很极端的方法。如果重载全局

版本，那么缺省版本就完全不能被访问——甚至在这个重载定义里也不能调用它们。

重载的new必须有一个size_t参数。这个参数由编译器产生并传递给我们，它是要分配内存的对象的长度。必须返回一个指向等于这个长度（或大于这个长度，如果我们有这样做的原因）的对象的指针，或如果没有找到存储单元（在这种情况下，构造函数不被调用），返回一个0。然而如果找不到存储单元，不能仅仅返回0，还应该调用new-handler或进行异常处理，通知这里存在问题。

运算符new的返回值是一个void*，而不是指向任何特定类型的指针。它所做的是分配内存，而不是完成一个对象的建立——直到构造函数调用了才完成对象的创建，这是由编译器所确保的动作，不在我们的控制范围内。

运算符delete接受一个指向由运算符new分配的内存的void*。它是一个void*因为它是在调用析构函数后得到的指针。析构函数从存储单元里移去对象。运算符 delete的返回类型是void。

下面提供了一个如何重载全局new和delete的简单的例子：

```
//: GLOBLNEW.CPP -- Overload global new/delete
#include <stdio.h>
#include <stdlib.h>

void* operator new(size_t sz) {
  printf("operator new: %d bytes\n", sz);
  void* m = malloc(sz);
  if(!m) puts("out of memory");
  return m;
}

void operator delete(void* m) {
  puts("operator delete");
  free(m);
}

class s {
  int i[100];
public:
  s() { puts("s::s()"); }
  ~s() { puts("s::~s()"); }
};

main() {
  puts("creating & destroying an int");
  int* p = new int(47);
  delete p;
  puts("creating & destroying an s");
  s* S = new s;
  delete S;
  puts("creating & destroying s[3]");
  s* SA = new s[3];
  delete []SA;
}
```

这里可以看到重载new和delete的一般形式。为了实现内存分配器，使用了标准 C库函数malloc( )和free( )（可能缺省的new和delete也使用这些函数）。它们还打印出了它们正在做什么的信息。注意，这里使用printf( )和puts( )而不是iostreams。当创建了一个iostream对象时（像全局的cin、cout和cerr），它们调用new去分配内存。用printf( )不会进入死锁状态，因为它不调用new来初始化本身。

在main( )里，创建内部数据类型的对象以证明在这种情况下重载的 new和delete也被调用。然后创建一个类型s的单个对象，接着创建一个数组。对于数组，我们可以看到需要额外的内存用于存放数组对象数量的信息。在所有情况里，都是使用全局重载版本的 new和delete。

12.5.2 为一个类重载new和delete

为一个类重载new和delete时，不必明说是static，我们仍是在创建static成员函数。它的语法也和重载任何其他运算符一样。当编译器看到使用 new创建类对象时，它选择成员版本运算符new而不是全局版本的new。但全局版本的new和delete为所有其他类型对象使用（除非它们有自己的new和delete）。

在下面的例子里我们为类framis创建了一个非常简单的内存分配系统。程序开始时在静态数据区域内留出一块存储单元。这块内存是用于 framis类型对象分配的内存空间。为了决定哪块存储单元已被使用，这里使用了一个字节(bytes)数组，一个字节(byte)代表一块存储单元。

```
//: FRAMIS.CPP -- Local overloaded new & delete
#include <stddef.h> // Size_t
#include <fstream.h>
ofstream out("framis.out");

class framis {
  char c[10];
  static unsigned char pool[];
  static unsigned char alloc_map[]; // Alloc map
public:
  enum { psize = 100 };  // # of framis allowed
  framis() { out << "framis()\n"; }
  ~framis() { out << "~framis() ... "; }
  void* operator new(size_t);
  void operator delete(void*);
};
unsigned char framis::pool[psize * sizeof(framis)];
unsigned char framis::alloc_map[psize] = {0};

// Size is ignored -- assume a framis object
void* framis::operator new(size_t) {
  for(int i = 0; i < psize; i++)
    if(!alloc_map[i]) {
      out << "using block " << i << " ... ";
      alloc_map[i] = 1; // Mark it used
      return pool + (i * sizeof(framis));
```

```
    }
  out << "out of memory" << endl;
  return 0;
}

void framis::operator delete(void* m) {
  if(!m) return; // Check for null pointer
  // Assume it was created in the pool
  // Calculate which block number it is:
  unsigned long block = (unsigned long)m
    - (unsigned long)pool;
  block /= sizeof(framis);
  out << "freeing block " << block << endl;
  // Mark it free:
  alloc_map[block] = 0;
}

main() {
  framis* f[framis::psize];
  for(int i = 0; i < framis::psize; i++)
    f[i] = new framis;
  new framis; // Out of memory
  delete f[10];
  f[10] = 0;
  // Use released memory:
  framis* x = new framis;
  delete x;
  for(int j = 0; j < framis::psize; j++)
    delete f[j]; // Delete f[10] OK
}
```

通过分配一个足够大的数组来保存 psize 个 framis对象的方法来为framis堆创建一块内存。这个内存分配图是psize个字节，所以每一块内存对应一个字节。使用设置首元素为 0的集合初始化技巧，编译器能够自动地初始化其余的元素，来使内存分配图里的所有字节都初始化为 0。

局部运算符new和全局运算符new具有相同的形式。它所做的是通过对内存分配图进行搜索来找到为0的字节，然后设置该字节为1，以此声明这块存储单元已经被分配并返回这个存储单元的地址。如果它找不到内存，将发送一个消息并返回 0（注意new-handler没有被调用，也没报告异常，这是因为当我们用完了内存时，行为在我们的控制之下）。因为没有涉及全局运算符new和delete，所以在这个例子里使用iostreams是可行的。

运算符delete假设framis的地址是在这个堆里创建的。这是一个公平的假设，因为无论何时在堆上创建单个framis对象 —— 不是一个数组，都将调用局部运算符new。全局版本的new在数组情况下使用。所以用户偶尔会在没有用空方括号语法来声明数组被消除的情况下调用运算符delete，这可能会引起问题。因为用户也可能删除在栈上创建的指向对象的指针。如果我们不想这样的事情发生，应该加入一行代码以确保地址是在这个堆内并是在正确的地址范围内。

运算符delete计算这个指针代表这个堆里的哪一块内存，然后将代表这块内存的分配图的标志设置为0来声明这块内存已经被释放。

在main( )里，动态地分配了很多framis对象以使内存用完，这样就能检查地址用完时的行为。然后释放一个对象，再创建一个对象来显示那个释放了的内存被重新使用了。

因为这个内存分配方案是针对framis对象的，所以可能比用缺省的new和delete针对一般目的内存分配方案效率要高一些。

12.5.3 为数组重载new和delete

如果为一个类重载了运算符new和delete，那么无论何时创建这个类的一个对象都将调用这些运算符。但如果为这些对象创建一个数组时，将调用全局运算符 new( )立即为这个数组分配足够的内存。全局运算符 delete( )被调用来释放这块内存。可以通过为那个类重载数组版本的运算符new[]和delete[]来控制对象数组的内存分配。这里提供了一个显示两个不同版本被调用的例子：

```
//: NEWARRY.CPP -- Operator new for arrays
#include <new.h> // Size_t definition
#include <fstream.h>
ofstream trace("newarry.out");

class widget {
  int i[10];
public:
  widget() { trace << "*"; }
  ~widget() { trace << "~"; }
  void* operator new(size_t sz) {
    trace << "widget::new: "
          << sz << " bytes" << endl;
    return ::new char[sz];
  }
  void operator delete(void* p) {
    trace << "widget::delete" << endl;
    ::delete []p;
  }
  void* operator new[](size_t sz) {
    trace << "widget::new[]: "
          << sz << " bytes" << endl;
    return ::new char[sz];
  }
  void operator delete[](void* p) {
    trace << "widget::delete[]" << endl;
    ::delete []p;
  }
};

main() {
```

```
    trace << "new widget" << endl;
    widget* w = new widget;
    trace << "\ndelete widget" << endl;
    delete w;
    trace << "\nnew widget[25]" << endl;
    widget* wa = new widget[25];
    trace << "\ndelete []widget" << endl;
    delete []wa;
};
```

这里，全局版本的new和delete被调用，除了加入了跟踪信息以外，它们和未重载版本 new和delete的效果是一样的。当然，我们可以在重载的new和delete里使用想要的内存分配方案。

可以看到除了加了一个括号外，数组版本的new和delete与单个对象版本是一样的。在这两种情况下，要传递分配的对象内存大小。传递给数组版本的内存大小是整个数组的大小。应该记住重载运算符new唯一需要做的是返回指向一个足够大的内存的指针。虽然我们可以初始化那块内存，但通常这是构造函数的工作，构造函数将被编译器自动调用。

这里构造函数和析构函数只是打印出字符，所以我们可以看到它们已被调用。下面是这个跟踪文件的输出信息：

```
new widget
widget::new: 20 bytes
*
delete widget
~widget::delete

new widget[25]
widget::new[]: 504 bytes
*************************
delete []widget
~~~~~~~~~~~~~~~~~~~~~~~~~widget::delete[]
```

和预计的一样（这个机器为一个 int使用2个字节），创建一个对象需要 20个字节。运算符new被调用，然后是构造函数（由 *指出）。以一个相反的形式调用 delete使得析构函数被调用，然后是delete。

当创建一个widget对象数组时，可使用数组版本new。但注意，申请的长度比期望的大4个字节。这额外的4个字节是系统用来存放数组信息的，特别是数组中对象的数量。当用下面的表达式时，方括号就告诉编译器它是一个对象数组，所以编译器产生寻找数组中对象的数量的代码，然后多次调用析构函数。

```
delete []widget ;
```

我们可以看到，即使数组运算符new和delete只为整个数组调用一次，但对于数组中的每一个对象，缺省的构造函数和析构函数都被调用。

12.5.4 构造函数调用

```
foo* f = new foo ;
```

上面的表达式调用new分配一个foo长度大小的内存，然后在那个内存上调用 foo构造函数。

假设所有的安全措施都失败了并且运算符 new返回值是0，将会发生什么？在上述情况下，构造函数没有调用，所以虽然没有一个成功创建的对象，但至少我们也没有调用构造函数和传递给它一个0指针。下面是一个证明这一点的例子：

```
//: NOMEMORY.CPP -- Constructor isn't called
// If new returns 0
#include <iostream.h>
#include <new.h> // size_t definition
void my_new_handler() {
  cout << "new handler called" << endl;
}

class nomemory {
public:
  nomemory() {
    cout << "nomemory::nomemory()" << endl;
  }
  void* operator new(size_t sz) {
    cout << "nomemory::operator new" << endl;
    return 0; // "Out of memory"
  }
};

main() {
  set_new_handler(my_new_handler);
  nomemory* nm = new nomemory;
  cout << "nm = " << nm << endl;
}
```

当程序运行时，它仅打印来自运算符 new的信息。因为new返回0，所以构造函数没有被调用，当然它的信息不会打印出来。

12.5.5 对象放置

重载运算符new还有其他两个不常见的用途。

1) 可能想要在内存的指定位置上放置一个对象。这对于面向硬件的内嵌系统特别重要，在这个系统中，一个对象可能和一个特定的硬件是同义的。

2) 可能想在调用new时，可以从不同的内存分配器中选择。

这两种情形都用相同的机制解决：重载的运算符 new可以带多于一个的参数。像前面所看到的，第一个参数总是对象的长度，它是在内部计算出来的并由编译器传递。但其他参数可以由我们自己定义：一个放置对象的地址、一个对内存分配函数或对象的引用、或其他方便的任何设置。

起先在调用过程中传递额外的参数给运算符 new的方法看起来似乎有点古怪：在关键字new后是参数表（没有size_t参数，它由编译器处理），参数表后面是正在创建的对象的类名字。例如：

```
X* xp = new(a) X ;
```

将传递a作为第二个参数给运算符new。当然，这是在这个运算符new已经声明的情况下才有效的。

下面的例子显示了如何在一个特定的存储单元里放置一个对象。

```
//: PLACEMNT.CPP -- Placement with operator new
#include <stddef.h> // Size_t
#include <iostream.h>

class X {
  int i;
public:
  X(int I = 0) { i = I; }
  ~X() {
    cout << "X::~X()" << endl;
  }
  void* operator new(size_t, void* loc) {
    return loc;
  }
};
main() {
  int l[10];
  X* xp = new(l) X(47); // X at location l
  xp->X::~X(); // Explicit destructor call
  // ONLY use with placement!
}
```

注意运算符new仅返回传递给它的指针。因此，应该由调用者决定对象存放在哪里，决定作为new表达式的一部分的构造函数将在哪块内存上被调用。

销毁对象时将会出现两难选择的局面。因为仅有一个版本运算符 delete，所以没有办法说“对这个对象使用我的特殊内存释放器”。我们可以调用析构函数，但不能用动态内存机制释放内存，因为内存不是在堆上分配的。

答案是用非常特殊的语法：可以显式地调用析构函数。例如：

```
xp->X::~X( ) ; //explicit destructor call
```

这里会出现一个严厉的警告。一些人把这看作是在范围结束前的某一时刻销毁对象的一种方法，而不是调整范围，或想要这个对象的生命期在运行时确定时，更正确地使用动态对象创建的方法。如果用这种方法为在栈上创建的对象调用析构函数，将会出现严重的问题，因为析构函数在出范围时又会被调用一次。如果为在堆上创建的对象用这种方法调用析构函数，析构函数将被执行，但内存不释放，这可能是我们不希望的结果。用这种方法显式地调用析构函数，其实只有一个原因，即为运算符new 支持存放语法。

虽然这个例子仅显示一个附加的参数，但如果为了其他目的而增加更多的参数，也是可行的。

12.6 小结

在栈上创建自动对象既方便又理想，但为了解决一般程序问题，我们必须在程序执行的任

何时候，特别是需要对来自程序外部信息反应时，能够创建和销毁对象。虽然 C的动态内存分配可以从堆上得到内存，但它在C++上不易使用且不能够保证安全。使用new和delete进行动态对象创建，这已经成为C++语言的核心，所以可以像在栈上创建对象一样容易地在堆上创建对象，另外，还可以有很大的灵活性。如果new和delete不能满足要求，尤其是它们没有足够的效率时，程序员还可以改变它们的行为，也可以在内存用完时进行修改。（第17章讨论的异常处理也在这里使用了）

12.7 练习

1. 写一个有构造函数和析构函数的类。在构造函数和析构函数里通过 cout宣布自己。通过这个类自己证明new和delete总是调用构造函数和析构函数。用 new创建这个类的一个对象，用delete销毁它。在堆上创建和销毁这些对象的一个数组。

2. 创建一个pstash对象，并把练习1的对象用new创建填入。观察当这个对象出了范围和它的析构函数被调用时发生的情况。

3. 写一个有单个对象版本和数组版本的重载运算符 new和delete的类。演示这两个版本的工作情况。

4. 设计一个对FRAMIS.CPP进行测试的程序来显示定制的new和delete比全局的new和delete大约快多少。

第13章　继承和组合

C++最重要的性能之一是代码重用。但是，为了具有可进化性，我们应当能够做比拷贝代码更多的工作。

在C的方法中，这个问题未能得到很好的解决。而用 C++，可以用类的方法解决，通过创建新类重用代码，而不是从头创建它们，这样，我们可以使用其他人已经创建并调试过的类。

关键是使用类而不是更改已存在的代码。这一章将介绍两种完成这件事的方法。第一种方法是很直接的：简单地创建一个包含已存在的类对象的新类，这称为组合，因为这个新类是由已存在类的对象组合的。

第二种方法更巧妙，创建一个新类作为一个已存在类的类型，采取这个已存在类的形式，对它增加代码，但不修改它。这个有趣的活动被称为继承，其中大量的工作由编译器完成。继承是面向对象程序设计的基石，并且还有另外的含义，将在下一章中探讨。

对于组合和继承（感觉上，它们都是由已存在的类型产生新类型的方法），它们在语法上和行为上是类似的。这一章中，读者将学习这些代码重用机制。

13.1　组合语法

实际上，我们一直都在用组合创建类，只不过我们是在用内部数据类型组合新类。其实使用用户定义类型组合新类同样很容易。

考虑下面这个在某种意义上很有价值的类：

```
//: USEFUL.H -- A class to reuse
#ifndef USEFUL_H_
#define USEFUL_H_

class X {
  int i;
  enum { factor = 11 };
public:
  X() { i = 0; }
  void set(int I) { i = I; }
  int read() const { return i; }
  int permute() { return i = i * factor; }
};
#endif // USEFUL_H_
```

在这个类中，数值成员是私有的，所以对于将类型 X 的一个对象作为公共对象嵌入到一个新类内部，是绝对安全的。

```
//: COMPOSE.CPP -- Reuse code with composition
#include "..\12\useful.h"

class Y {
```

```
  int i;
public:
  X x; // Embedded object
  Y() { i = 0; }
  void f(int I) { i = I; }
  int g() const { return i; }
};

main() {
  Y y;
  y.f(47);
  y.x.set(37); // Access the embedded object
}
```

访问嵌入对象（称为子对象）的成员函数只须再一次选择成员。

如果嵌入的对象是private的，可能更具一般性，这样，它们就变成了内部实现的一部分（这意味着，如果我们愿意，我们可以改变这个实现）。而对于新类的public接口函数，包含对嵌入对象的使用，但不必模仿这个嵌入对象的接口。

```
//: COMPOSE2.CPP -- Private embedded objects
#include "..\12\useful.h"
class Y {
  int i;
  X x; // Embedded object
public:
  Y() { i = 0; }
  void f(int I) { i = I; x.set(I); }
  int g() const { return i * x.read(); }
  void permute() { x.permute(); }
};

main() {
  Y y;
  y.f(47);
  y.permute();
}
```

这里，permute()函数的执行调用了X的接口，而X的其他成员函数也在Y的成员函数中被调用。

13.2 继承语法

组合的语法是明显的，而完成继承，则有新的不同的形式。

继承也就是说“这个新的类像那个老的类”。通过给出这个类的名字，在代码中声明继承，在这个类体的开括号前面，加一冒号和基类名（或加多个类，对于多重继承）。这样做，就自动地得到了基类中的所有数据成员和成员函数。下面是一个例子：

在Y中，我们可以看到继承，它意味着 Y 将包含 X 中的所有数据成员和成员函数。实际

```
//: INHERIT.CPP -- Simple inheritance
#include "..\12\useful.h"
#include <iostream.h>
class Y : public X {
  int i; // Different from X's i
public:
  Y() { i = 0; }
  int change() {
    i = permute(); // Different name call
    return i;
  }
  void set(int I) {
    i = I;
    X::set(I); // Same-name function call
  }
};

main() {
  cout << "sizeof(X) = " << sizeof(X) << endl;
  cout << "sizeof(Y) = "
       << sizeof(Y) << endl;
  Y D;
  D.change();
  // X function interface comes through:
  D.read();
  D.permute();
  // Redefined functions hide base versions:
  D.set(12);
}
```

上，Y包含了 X 的一个子对象，就像在 Y 中创建 X 的一个成员对象，而不从 X 继承一样。无论成员对象还是基类存储，都被认为是子对象。在main()，可以看到这些数据成员已被加入了，因为sizeof(Y）是sizeof(X)的 2 倍大。

我们将注意到，基类由public处理，否则，在继承中，所有被继承的东西都是 private，也就是说在基类中的所有public成员在派生类中都是private。这当然不是所希望的，希望的结果是保持基类中的public成员在派生类中也是public。这可在继承期间用关键字public做到。

在change()中，基类permute()函数被调用，派生类对所有的public基类函数都有直接的访问权。

在派生类中的set()函数重定义了基类中的set()函数。这就是，如果对于类型 Y 的对象调用函数read()和permute()，得到的是这些函数的基类版本（可以在 main()中看到表现）。但如果对于对象 Y 调用 set()，得到的是重定义的版本。这意味着，如果我们不喜欢在继承中得到某个函数的基类版本，可以改变它（还能够增加全新的函数，例如 change()）。

然而，当重定义函数时，我们可能希望仍然保留基类版本。如果简单地调用 set()，得到的是这个函数的本地版本（一个递归函数调用）。为了调用基类版本，我们必须用准确名，即使

用基类名和范围分解运算符。

13.3 构造函数的初始化表达式表

我们已经看到，在C++中保证合适的初始化表达式多么重要，在组合和继承中，也是一样。当创建一个对象时，必须保证编译器调用所有子对象的构造函数。到目前为止，例子中的所有子对象都有缺省的构造函数，编译器可以自动调用它们。但是，如果子对象没有缺省构造函数或如果我们想改变某个构造函数的缺省参数，情况会怎么样呢？这是一个问题，因为这个新类的构造函数不能保证访问它的子对象的private数据成员，所以不能直接地对它们初始化。

解决办法很简单：对于子对象调用构造函数，C++为此提供了专门的语法，即构造函数的初始化表达式表。构造函数的初始化表达式表的形式模仿继承活动。对于继承，我们在冒号之后和这个类体的左括号之前放置基类。而在构造函数的初始化表达式表中，我们可以将对子对象构造函数的调用语句放在构造函数参数表和冒号之后，在函数体的开括号之前。对于从 bar 继承来的类 foo，如果 bar 有一个取单个int参数的构造函数，则表示为

```
foo::foo(int i) : bar(i) { //...
```

13.3.1 成员对象初始化

当使用组合时，对于成员对象初始化使用相同的语法是不可行的。对于组合，给出对象的名字而不是类名。如果在初始化表达式表中有多于一个构造函数调用，应当用逗号隔开：

```
foo2:foo2(int I) : bar(i), memb(i+1) { // ...
```

这是类 foo2 构造函数的开头，它是从 bar 继承来的，包含称为 memb 的成员对象。注意，当我们在这个构造函数的初始化表达式表中能看到基类的类型时，只能看到成员对象的标识符。

13.3.2 在初始化表达式表中的内置类型

构造函数的初始化表达式表允许我们显式地调用成员对象的构造函数。事实上，这里没有其它方法调用那些构造函数。主要思想是，在进入新类的构造函数体之前调用所有的构造函数。这样，对子对象的成员函数所做的任何调用都已经转到了这个被初始化的对象中。没有对所有的成员对象和基类对象的构造函数调用，就没有办法进入这个构造函数的左括号，即便是编译器也必须对缺省构造函数做隐藏调用。这是 C++进一步的强制，以保证没有调用它的构造函数就没有对象（或对象的部分）能进入第一道门。

所有的成员对象在构造函数的左括号之前被初始化的思想是编程时一个方便的辅助方法。一旦遇到左括号，我们能假设所有的子对象已被适当地初始化了，并集中精力在希望该构造函数完成的特殊任务上。然而，这里还有一个问题：内置类型的嵌入对象如何？它没有构造函数吗？

为了让语法一致，允许对待内置类型就像对待有单个构造函数的对象一样，它取单个参数：这个参数与我们正在初始化的变量类型相同。这样，我们就可以写：

```
class X {
  int i;
  float f;
  char c;
  char* s;
public:
```

```
  X() : i(7), f(1.4), c('x'), s("howdy") {}
  // ...
```

这些“伪构造函数调用”是为了完成简单的赋值。它是传统的技术和好的编码风格，所以常常能看到它被使用。

甚至在类之外创建这种类型的变量时，我们也可能用伪构造函数语法：

```
int i(100);
```

这使得内置类型的效果更像对象。

记住，这些并不真的是构造函数，特别是，如果没有显式地进行伪构造函数调用，就不会进行初始化。

13.4 组合和继承的联合

当然，我们还可以把两者放在一起使用。下面的例子中这个更复杂类的创建就使用了继承和组合两种方法。

```
//: COMBINED.CPP -- Inheritance & composition

class A {
  int i;
public:
  A(int I) { i = I; }
  ~A() {}
  void f() const {}
};

class B {
  int i;
public:
  B(int I) { i = I; }
  ~B() {}
  void f() const {}
};

class C : public B {
  A a;
public:
  C(int I) : B(I), a(I) {}
  ~C() {} // Calls ~A() and ~B()
  void f() const {  // Redefinition
    a.f();
    B::f();
  }
};

main() {
  C c(47);
}
```

C 继承 B 并且有一个成员对象（这是类 A的对象）。我们可以看到，构造函数的初始化表达式表中调用了基类构造函数和成员对象构造函数。

函数C::f() 重定义了它所继承的B::f()，并且还调用基类版本。另外，它还调用了a.f()。注意，只能在继承期间重定义函数。通过成员对象，只能操作这个对象的公共接口，而不能重定义它。另外，如果C::f() 还没有被定义，则对类型C 的一个对象调用 f() 不会调用 a.f()，而是调用 B::f()。

• 自动析构函数调用

虽然常常需要在初始化表达式表中做显式构造函数调用，但我们决不需要做显式的析构函数调用，因为对于任何类型只有一个析构函数，并且它并不取任何参数。然而，编译器仍保证所有的析构函数被调用，这意味着，在整个层次中的所有析构函数，从最底层的析构函数开始调用，一直到根层。

构造函数和析构函数与众不同之处在于每一层函数都被调用，这是值得强调的。然而对于一般的成员函数，只是这个函数被调用，而不是任意基类版本被调用。如果还想调用一般成员函数的基类版本，必须显式地调用。

13.4.1 构造函数和析构函数的次序

当一个对象有许多子对象时，知道构造函数和析构函数的调用次序是有趣的。下面的例子明显表明它如何工作：

```
//: ORDER.CPP -- Constructor/destructor order
#include <fstream.h>
ofstream out("order.out");

#define CLASS(ID) class ID { \
public: \
  ID(int) { out << #ID " constructor\n"; } \
  ~ID() { out << #ID " destructor\n"; } \
};

CLASS(base1);
CLASS(member1);
CLASS(member2);
CLASS(member3);
CLASS(member4);

class derived1 : public base1 {
  member1 m1;
  member2 m2;
public:
  derived1(int) : m2(1), m1(2), base1(3) {
    out << "derived1 constructor\n";
  }
  ~derived1() {
    out << "derived1 destructor\n";
  }
};
```

```
class derived2 : public derived1 {
  member3 m3;
  member4 m4;
public:
  derived2() : m3(1), derived1(2), m4(3) {
    out << "derived2 constructor\n";
  }
  ~derived2() {
    out << "derived2 destructor\n";
  }
};

main() { derived2 d2; }
```

首先，创建 ofstream 对象，以发送所有输出到一个文件中。为了在书中少敲一些字符也为了演示一种宏技术（这个技术将在第 18章中被一个更好的技术代替），这里使用了宏以建立一些类（这些类将被用于继承和组合）。每个构造函数和析构函数向这个跟踪文件报告它们自己的行动。注意，这些是构造函数，而不是缺省构造函数，它们每一个都有一整型参数。这个参数本身没有标识符，它的唯一的任务就是强迫在初始化表达式表中显式调用这些构造函数。（消除标识符防止编译器警告信息）这个程序的输出是：

```
base1 constructor
member1 constructor
member2 constructor
derived1 constructor
member3 constructor
member4 constructor
derived2 constructor
derived2 destructor
member4 destructor
member3 destructor
derived1 destructor
member2 destructor
member1 destructor
base1 destructor
```

可以看出，构造在类层次的最根处开始，而在每一层，首先调用基类构造函数，然后调用成员对象构造函数。调用析构函数则严格按照构造函数相反的次序 —— 这是很重要的，因为要考虑潜在的相关性。另一有趣的是，对于成员对象，构造函数调用的次序完全不受在构造函数的初始化表达式表中次序的影响。该次序是由成员对象在类中声明的次序所决定的。如果能通过构造函数的初始化表达式表改变构造函数调用次序，那么就会对两个不同的构造函数有二种不同的调用顺序。而析构函数不可能知道如何为析构函数相应地反转调用次序，这就引起了相关性问题。

13.4.2 名字隐藏

如果在基类中有一个函数名被重载几次，在派生类中重定义这个函数名会掩盖所有基类版

本，这也就是说，它们在派生类中变得不再可用。

```
//: HIDE.CPP -- Name hiding during inheritance

class homer {
public:
  int doh(int) const { return 1; }
  char doh(char) const { return 'd';}
  float doh(float) const { return 1.0; }
};

class bart : public homer {
public:
  class milhouse {};
  void doh(milhouse) const {}
};

main() {
  bart b;
//! b.doh(1); // Error
//! b.doh('x'); // Error
//! b.doh(1.0); // Error
}
```

因为 bart 重定义了 doh()，这些基类版本中没有一个是对于 bart 对象可调用的。这时,编译器试图变换参数成为一个 milhouse 对象，并报告出错，因为它不能找到这样的变换。正如在下面章节中会看到的，更普遍的方法是用与在基类中严格相同的符号重定义函数并且返回类型也与基类中的相同。

13.4.3 非自动继承的函数

不是所有的函数都能自动地从基类继承到派生类中的。构造函数和析构函数是用来处理对象的创建和析构的，它们只知道对在它们的特殊层次的对象做什么。所以，在整个层次中的所有的构造函数和析构函数都必须被调用，也就是说，构造函数和析构函数不能被继承。

另外，operator= 也不能被继承，因为它完成类似于构造函数的活动。这就是说，尽管我们知道如何由等号右边的对象初始化左边的对象的所有成员，但这并不意味着这个初始化在继承后仍有意义。在继承过程中，如果我们不亲自创建这些函数，编译器就综合它们。（通过构造函数，我们不能对缺省构造函数和被自动创建的拷贝构造函数创建任何构造函数）这在第 11 章中已经简要地讲过了。被综合的构造函数使用成员方式的初始化，而被综合的 operator= 使用成员方式的赋值。这是由编译器创建而不是继承的函数的例子。

```
//: NINHERIT.CPP -- Non-inherited functions
#include <fstream.h>
ofstream out("ninherit.out");

class root {
public:
```

```
  root() { out << "root()\n"; }
  root(root&) { out << "root(root&)\n"; }
  root(int) { out << "root(int)\n"; }
  root& operator=(const root&) {
    out << "root::operator=()\n";
    return *this;
  }
  class other {};
  operator other() const {
    out << "root::operator other()\n";
    return other();
  }
  ~root() { out << "~root()\n"; }
};

class derived : public root {};

void f(root::other) {}

main() {
  derived d1;  // Default constructor
  derived d2 = d1; // Copy-constructor
//! derived d3(1); // Error: no int constructor
  d1 = d2; // Operator= not inherited
  f(d1); // Type-conversion IS inherited
}
```

所有的构造函数和 operator= 都自我宣布，所以我们能知道编译器何时使用它们。另外，operator other() 完成自动类型变换，从 root 对象到被嵌入的类 other 的对象。类 derived 直接从 root 继承，并没有创建函数（观察编译器如何反应）。函数 f() 取一个 other 对象以测试这个自动类型变换函数。

在 main() 中，创建缺省构造函数和拷贝构造函数，调用 root 版本作为构造函数调用继承的一部分，尽管这看上去像是继承，但新的构造函数实际上是创建的。正如我们所预料的，自动创建带参数的构造函数是不可能的，因为这样太依赖编译器的直觉。

在 derived 中，operator=( ) 也被综合为新函数，使用成员函数赋值，因为这个函数在新类中不显式地写出。

关于处理对象创建的重写函数的所有这些原则，我们也许会觉得奇怪，为什么自动类型变换运算也能被继承。但其实这不足为奇——如果在 root 中有足够的块建立一个 other 对象，那么从 root 派生出的任何东西中，这些块仍在原地，类型变换当然也就仍然有效。（尽管实际上我们可能想重定义它）

13.5 组合与继承的选择

无论组合还是继承都能把子对象放在新类型中。两者都使用构造函数的初始化表达式表去构造这些子对象。现在我们可能会奇怪，这两者之间到底有什么不同？该如何选择？

组合通常在希望新类内部有已存在类性能时使用，而不希望已存在类作为它的接口。这就是说，嵌入一个计划用于实现新类性能的对象，而新类的用户看到的是新定义的接口而不是来自老类的接口。为此，在新类的内部嵌入已存在类的 private 对象。

有时，允许类用户直接访问新类的组合是有意义的，这就让成员对象是 public。成员函数隐藏它们自己的实现，所以，当用户知道我们正在装配一组零件并且使得接口对他们来说更容易理解时，这样会安全的。Car 对象是一个很好的例子：

```
//: CAR.CPP -- Public composition

class engine {
public:
  void start() const {}
  void rev() const {}
  void stop() const {}
};

class wheel {
public:
  void inflate(int psi) const {}
};

class window {
public:
  void rollup() const {}
  void rolldown() const {}
};

class door {
public:
  window Window;
  void open() const {}
  void close() const {}
};

class car {
public:
  engine Engine;
  wheel Wheel[4];
  door left, right; // 2-door
};

main() {
  car Car;
  Car.left.Window.rollup();
  Car.Wheel[0].inflate(72);
}
```

因为小汽车的组合是分析这个问题的一部分（不是基本设计的部分），所以让成员是公共的有助于客户程序员理解如何使用这个类，而且能使类的创建者有更小的代码复杂性。

稍加思考就会看到，用车辆对象组合小汽车是无意义的——小汽车不能包含车辆，它本身就是一种车辆。这种is-a关系用继承表达，而 has-a 关系用组合表达。

13.5.1 子类型设置

现在假设想创建包含 ifstream 对象的一个类，它不仅打开一个文件，而且还保存文件名。这时我们可以使用组合并把 ifstream 及 strstream 都嵌入这个新类中：

```
//: FNAME1.CPP -- An fstream with a file name
#include <fstream.h>
#include <strstrea.h>
#include "..\allege.h"

class fname1 {
  ifstream File;
  enum { bsize = 100 };
  char buf[bsize];
  ostrstream Name;
  int nameset;
public:
  fname1() : Name(buf, bsize), nameset(0) {}
  fname1(const char* filename)
    : File(filename), Name(buf, bsize) {
      allegefile(File);
      Name << filename << ends;
      nameset = 1;
  }
  const char* name() const { return buf; }
  void name(const char* newname) {
    if(nameset) return; // Don't overwrite
    Name << newname << ends;
    nameset = 1;
  }
  operator ifstream&() { return File; }
};
main() {
  fname1 file("fname1.cpp");
  cout << file.name() << endl;
  // Error: rdbuf() not a member:
//!  cout << file.rdbuf() << endl;
}
```

然而这里存在一个这样的问题：我们也许想通过包含一个从 fname1 到ifstream &的自动类型转换运算，在任何使用 ifstream 的地方都使用 fname1 对象，但在 main 中，

```
cout<<file.rdbuf()<<endl;
```

这一行不能编译，因为自动类型转换只发生在函数调用中，而不在成员选择期间。所以，此时这个方法不行。

第二个方法是对 fname1 增加 rdbuf() 定义：

```
filebuf * rdbuf()  {return File.rdbuf();}
```

如果只有很少的函数从 ifsream 类中拿来，这是可行的。在这种情况下，我们只是使用了这个类的一部分，并且组合是适用的。

但是，如果我们希望这个类的东西都进来，应该做什么呢？这称为子类型设置，因为正在由一个已存在的类做一个新类，并且希望这个新类与已存在的类有严格相同的接口（希望增加任何我们想要加入的其他成员函数），所以能在已经用过这个已存在类的任何地方使用这个新类，这就是必须使用继承的地方。我们可以看到，子类型设置很好地解决了先前例子中的问题。

```
//: FNAME2.CPP -- Subtyping solves the problem
#include <fstream.h>
#include <strstrea.h>
#include "..\allege.h"
class fname2 : public ifstream {
  enum { bsize = 100 };
  char buf[bsize];
  ostrstream Name;
  int nameset;
public:
  fname2() : Name(buf, bsize), nameset(0) {}
  fname2(const char* filename)
    : ifstream(filename), Name(buf, bsize) {
      Name << filename << ends;
      nameset = 1;
  }
  const char* name() const { return buf; }
  void name(const char* newname) {
    if(nameset) return; // Don't overwrite
    Name << newname << ends;
    nameset = 1;
  }
};

main() {
  fname2 file("fname2.cpp");
  allegefile(file);
  cout << "name: " << file.name() << endl;
  const bsize = 100;
  char buf[bsize];
  file.getline(buf, bsize); // This works too!
  file.seekg(-200, ios::end);
  cout << file.rdbuf() << endl;
}
```

现在，能与ofstream 对象一起工作的任何成员函数也能与 fname2 对象一起工作。这是因为，fname2 是ofstream的一个类型。不是简单地包含。这是非常重要的问题，将在本章最后和在第14章中讨论。

13.5.2 专门化

继承也就是取一个已存在的类，并制作它的一个专门的版本。通常，这意味着取一个一般目的的类并为了特殊的需要对它专门化。

例如，考虑前面章中的 stack 类，与这个类有关的问题之一是必须每次完成计算，从包容器中引出一个指针。这不仅乏味，而且不安全——我们能让这个指针指向所希望的任何地方。

较好的方法是使用继承来专门化这个一般的 stack 类。这里有一个例子，它使用来自前一章的类。

```
//: INHSTAK.CPP -- Specializing the stack class
#include "..\11\stack11.h"
#include "..\11\strings.h"
#include <fstream.h>
#include "..\allege.h"

class Stringlist : public stack {
public:
  void push(String* str) {
    stack::push(str);
  }
  String* peek() const {
    return (String*)stack::peek();
  }
  String* pop() {
    return (String*)stack::pop();
  }
};

main() {
  ifstream file("inhlist.cpp");
  allegefile(file);
  const bufsize = 100;
  char buf[bufsize];
  Stringlist textlines;
  while(file.getline(buf,bufsize))
    textlines.push(String::make(buf));
  String* s;
  while((s = textlines.pop()) != 0) // No cast!
    cout << *s << endl;
}
```

两个头文件STRINGS.H（第12.2.1节）和 STACK11.H （第12.2.3章）都从第12章引来

（STACK11.OBJ 文件也必须连进来。）

stringlist专门化stack ，所以push()只接受string指针。在此之前，stack会接受void指针，所以用户没有类型检查以确保插入合适的指针。另外， peek()和pop()现在返回string指针，而不返回void指针，所以使用这个指针时不必映射。

令人惊奇的是，额外的类型安全性检查是无需代价的！编译器给出额外的类型信息，这些只在编译时使用，而这些函数是内联的，并不产生另外的代码。

不幸的是，继承并不能解决所有这些与包容器类有关的问题，析构函数仍然引起麻烦。我们也许会记得，在第12章中，stack::~stack()析构函数遍历整个表，并对所有的指针调用 delete。问题是，对于void指针调用delete，它只释放这块内存而不调用析构函数（因为 void没有类型信息）。可以创建一个stringlist::~stringlist()析构函数，它能遍历这个表并对所有在表中的 string指针调用delete，这样，问题就解决了。条件是：

1) stack数据成员被定义为protected，使得这个stringlist析构函数能访问它们。（protected在本章稍后介绍。）

2) 移去stack基类析构函数，使得这块内存不会两次被释放。

3) 不执行更多的继承，否则会再次面临相同的两难境地：要么调用多重析构函数，要么进行不正确的析构函数调用（可能包含string对象而不是从stringlist派生出的类）。

这个问题将在下一章重述，但直到第15章介绍了模板后才能得到完全解决。

13.5.3 私有继承

通过在基类表中去掉 public 或者通过显式地声明private，可以私有地继承基类（后者可能是更好的策略，因为可以让用户明白它的含义）。当私有继承时，创建的新类有基类的所有数据和功能，但这些功能是隐藏的，所以它只是内部实现部分。该类的用户访问不到这些内部功能，并且一个对象不被看作这个基类的成员（如在第 13.5.1节中的FNAME2.CPP中的）。

我们可能奇怪， private 继承的目的是什么，因为在这个新类中选择创建一个 private对象似乎更合适。将private继承包含在该语言中只是为了语言的完整性。但是，如果没有其他理由，则应当减少混淆，所以通常建议用 private成员而不是private继承。然而，这里可能偶然有这种情况，即可能想产生像基类接口一样的接口，而不允许处理该对象像处理基类对象一样。private继承提供了这个功能。

• 对私有继承成员公有化

当私有继承时，基类的所有 public成员都变成了private。如果希望它们中的任何一个是可视的，只要用派生类的public选项声明它们的名字即可。

```
//: PRIVINH.CPP -- Private inheritance

class base1 {
public:
  char f() const { return 'a'; }
  int g() const { return 2; }
  float h() const { return 3.0; }
};

class derived : base1 { // Private inheritance
public:
```

```
  base1::f; // Name publicizes member
  base1::h;
};

main() {
  derived d;
  d.f();
  d.h();
//! d.g(); // Error -- private function
}
```

这样，如果想要隐藏这个类的基类部分的功能，则private继承是有用的。

在我们使用private继承取代对象成员之前，应当注意到，当与运行类型标识相连时，私有继承有特定的复杂性。（第18章内容。）

13.6 保护

关键字protected对于继承有特殊的意义。在理想世界中，private成员总是严格私有的，但在实际项目中，有时希望某些东西隐藏起来，但仍允许其派生类的成员访问。于是关键字protected派上了用场。它的意思是：“就这个类的用户而言，它是private的，但它可被从这个类继承来的任何类使用。”

数据成员最好是private，因为我们应该保留改变内部实现的权利。然后我们才能通过保护成员函数控制对该类的继承者的访问。

```
//: PROTECT.CPP -- The protected keyword
#include <fstream.h>

class base {
  int i;
protected:
  int read() const { return i; }
  void set(int I) { i = I; }
public:
  base(int I = 0) : i(I) {}
  int value(int m) const { return m*i; }
};

class derived : public base {
  int j;
public:
  derived(int J = 0) : j(J) {}
  void change(int x) { set(x); }
};

main() {}
```

在附录C中的SSHAPE例子中，我们可以看到需要protected的很好的例子。

被保护的继承

继承时，基类缺省为private，这意味着所有public成员函数对于新类的用户是private的。通常我们都会让继承public，从而使得基类的接口也是派生类的接口。然而在继承期间，也可以使用protected关键字。

被保护的派生意味着对其他类来“照此实现”，但对派生类和友元是“is-a”。它是不常用的，它的存在只是为了语言的完整性。

13.7 多重继承

既然我们已可以从一个类继承，那么我们也就应该能同时从多个类继承。实际上这是可以做到的，但是否它象设计部分一样有意义仍是一个有争议的话题。不过有一点是可以肯定的：直到我们已经很好地学会程序设计并完全理解这门语言时，我们才能试着用它。这时，我们大概会认识到，不管我们如何认为我们必须用多重继承，我们总是能通过单重继承来完成。

起初，多重继承似乎很简单，在继承期间，只需在基类表中增加多个类，用逗号隔开。然而，多重继承有很多含糊的可能性，这就是为什么第16章要讨论这一主题的原因。

13.8 渐增式开发

继承的优点之一是它支持渐增式开发，它允许我们在已存在的代码中引进新代码，而不会给原代码带来错误，即使产生了错误，这个错误也只与新代码有关。也就是说当我们继承已存在的功能类并对其增加数据成员和成员函数（并重定义已存在的成员函数）时，已存在类的代码并不会被改变，更不会产生错误。

如果错误出现，我们就会知道它肯定是在我们的新派生代码中。相对于修改已存在代码体的做法来说，这些新代码很短也很容易读。

相当奇怪的是，这些类如何清楚地被隔离。为了重用这些代码，甚至不需要这些成员函数的源代码，只需要表示类的头文件和目标文件或带有已编译成员函数的库文件。（对于继承和组合都是这样。）

认识到程序开发是一个渐增过程，就象人的学习过程一样，这是很重要的。我们能做尽可能多的分析，但当开始一个项目时，我们仍不可能知道所有的答案。如果开始把项目作为一个有机的、可进化的生物来“培养”，而不是完全一次性的构造它，使之像一个玻璃盒子式的摩天大楼，那么我们就会获得更大的成功和更直接的反馈。

虽然继承对于实验是有用的技术，但在事情稳定之后，我们需要用新眼光重新审视一下我们的类层次，把它看成可感知的结构。记住，继承首先表示一种关系，其意为：“新类是老类的一个类型。”我们的程序不应当关心怎样摆布比特位，而应当关心如何创建和处理各类型的对象，以便用问题的术语表示模型。

13.9 向上映射

在这一章的前面，我们已经看到了由ofstream派生而来的类的对象如何有ofstream对象所有的特性和行为。在13.5.1节中FNAME2.CPP中，任何ofstream成员函数应当能被fname2对象调用。

继承的最重要的方面不是它为新类提供了成员函数，而在于它是基类与新类之间的关系描述：“新类是已存在类的一个类型”。

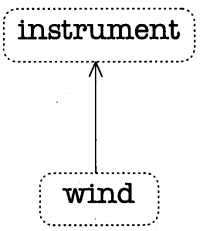
这个描述不仅仅是一种解释继承的方法——它直接由编译器支持。例如，考虑称为

instrument的基类（它表示乐器）和派生类wind，因为继承意味着在基类中的所有函数在派生类中也是可行的，可以发送给基类的消息也可以发送给这个派生类，所以，如果 instrument类有play()成员函数，那么wind 也有。这意味着，我们可以确切地说， wind是instrument 的一个类型。下面的例子表明编译器是如何支持这个概念的。

```
//: WIND.CPP -- Inheritance & upcasting
enum note { middleC, Csharp, Cflat }; // Etc.

class instrument {
public:
  void play(note) const {}
};

// Wind objects are instruments
// because they have the same interface:
class wind : public instrument {};

void tune(instrument& i) {
  // ...
  i.play(middleC);
}

main() {
  wind flute;
  tune(flute); // Upcasting
}
```

在这个例子中，有趣的是tune()函数，它接受instrument参数。然而，在main()中，tune()函数的调用却被传递了一个wind参数 。我们可能会感到奇怪， C++对于类型检查应该是非常严格的，而接受某个类型的函数为什么会这么容易地接受另一个类型。直到人们认识到 wind对象也是一个instrument对象，tune()函数能对instrument 调用，也能对wind调用时，才会恍然大悟。在tune()中，这些代码对instrument和从instrument派生来的任何类型都有效，这种将 wind的对象、引用或指针转变成instrument对象、引用或指针的活动称为向上映射。

13.9.1 为什么“向上映射”



这个术语引入的是有其历史原因的，而且它也与类继承图的传统画法有关：在顶部是根，向下长（当然我们可以用任何我们认为方便的方法画我们的图）。对于WIND.CCP的继承如右图

从派生类到基类的映射，在继承图中是上升的，所以一般称为向上映射。向上映射总是安全的。因为是从更专门的类型到更一般的类型——对于这个类接口可能出现的唯一的情况是它失去成员函数，不会增加成员函数。这就是编译器允许向上映射不需要显式地说明或做其他标记的原因。

• 向下映射

当然我们也可以实现向上映射的反转，称为向下映射，但是，这涉及到一个两难问题，这是第17章中讨论的主题。

13.9.2 组合与继承

确定应当用组合还是用继承，最清楚的方法之一是询问是否需要新类向上映射。在本章的前面，stack类通过继承被专门化，然而，stringlist对象仅作为string包容器，不需向上映射，所以更合适的方法可能是组合：

```
//: INHSTAK2.CPP -- Composition vs inheritance
#include "..\11\stack11.h"
#include "..\11\strings.h"
#include <fstream.h>
#include "..\allege.h"

class Stringlist {
  stack Stack; // Embed instead of inherit
public:
  void push(String* str) {
    Stack.push(str);
  }
  String* peek() const {
    return (String*)Stack.peek();
  }
  String* pop() {
    return (String*)Stack.pop();
  }
};

main() {
  ifstream file("inhlst2.cpp");
  allegefile(file);
  const bufsize = 100;
  char buf[bufsize];
  Stringlist textlines;
  while(file.getline(buf,bufsize))
    textlines.push(String::make(buf));
  String* s;
  while((s = textlines.pop()) != 0) // No cast!
    cout << *s << endl;
}
```

这个文件与INHSTACK.CPP(13.5.2节中)是一样的，只不过stack对象被嵌入在stringlist内，还有被嵌入对象调用的一些函数也被嵌入在 stringlist内。这里没有时间和空间的开销，因为其子类占用相同量的空间，而且所有另外的类型检查都发生在编译时。

我们也可以用private继承以表示“照此实现”。用以创建stringlist类的方法在这种情况下不是重要的——因为各种方法都能解决问题。然而，当可能存在多重继承时就要注意了，多重继承时可能被警告。在这种情况下，如果能发现一个类组合可以使用，那就不要用继承，因为这样可以消除对多重继承的需要。

13.9.3 指针和引用的向上映射

在WIND.CPP(13.9)中，向上映射发生在函数调用期间——在函数外的wind对象被引用并且变成一个在这个函数内的instrument的引用。

向上映射还能出现在对指针或引用简单赋值期间：

```
wind w;
instrument* ip = &w; // upcast
instrument& ir = w; // upcast
```

和函数调用一样，这两个例子都不要求显式地映射。

13.9.4 危机

当然，任何向上映射都会损失对象的类型信息，如果说

```
wind w;
instrument * ip = &w;
```

编译器只能把ip作为一个instrument指针处理。这就是说，它不知道ip实际上指向wind的对象。所以，当调用play()成员函数时，如果使用

```
ip->play(middleC);
```

编译器只知道它正在对于一个instrument指针调用play()，并调用instrument::play()的基本版本，而不是它应该做的调用wind::play()。这样将会得到不正确的结果。

这是一个重要的问题，将在下一章通过介绍面向对象编程的第三块基石：多态性（在C++中用virtual函数实现）来解决。

13.10 小结

继承和组合都允许由已存在的类型创建新类型，两者都是在新类型中嵌入已存在的类型的子对象。然而，当我们想重用原类型作为新类型的内部实现的话，我们最好用组合，如果我们不仅想重用这个内部实现而且还想重用原来接口的话那就用继承。如果派生类有基类的接口，它就能向上映射到这个基类，这一点对多态性很重要，这将在下一章中讲到。

虽然通过组合和继承进行代码重用对于快速项目开发有帮助，但通常我们会希望在允许其他程序员依据它开发程序之前重新设计类层次。

我们的类层次必须有这样的特性：它的每个类有专门的用途，不能太大（包含太多的功能不利于重用），也不能太小（太小如不对它本身增加功能就不能使用）。而且这些类应当容易重用。

13.11 练习

1. 修改CAR.CPP，使得它也从被称为vehicle的类继承，在vehicle中放置合适的成员函数（也就是说，补充一些成员函数）。对vehicle增加一个非缺省的构造函数，在car的构造函数内部必须调用它。

2. 创建两个类，A和B，带有能宣布自己的缺省构造函数。从A继承出一个新类，称为C，并且在C中创建B的一个成员对象，而不对C创建构造函数。创建类C的一个对象，观察结果。

3. 使用继承，专门化在第12章（PSTASH.H & PSTASH.CPP）中的pstash类，使得它接受和返回String指针。修改PSTEST.CPP并测试它。改变这个类使得pstash是一个成员对象。

4. 使用private和protected继承从基类创建两个新类。然后尝试向上映射这个派生类的对象成为基类。解释所发生的事情。

第14章　多态和虚函数

多态性（在C++中用虚函数实现）是面向对象程序设计语言继数据抽象和继承之后的第三个基本特征。

它提供了与具体实现相隔离的另一类接口，即把“ what ”从“ how ”分离开来。多态性提高了代码的组织性和可读性，同时也可使得程序具有可生长性，这个生长性不仅指在项目的最初创建期可以“ 生长 ”，而且希望项目具有新的性能时也能“ 生长 ”。

封装是通过特性和行为的组合来创建新数据类型的，通过让细节 private来使得接口与具体实现相隔离。这类机构对于有过程程序设计背景的人来说是非常有意义的。而虚函数则根据类型的不同来进行不同的隔离。上一章，我们已经看到，继承如何允许把对象作为它自己的类型或它的基类类型处理。这个能力很重要，因为它允许很多类型（从同一个基类派生的）被等价地看待就象它们是一个类型，允许同一段代码同样地工作在所有这些不同类型上。虚函数反映了一个类型与另一个类似类型之间的区别，只要这两个类型都是从同一个基类派生的。这种区别是通过其在基类中调用的函数的表现不同来反映的。

在这一章中，我们将从最基本的内容开始学习虚函数，为了简单起见，本章所用的例子经过简化，只保留了程序的虚拟性质。

• C++程序员的进步

C程序员似乎可以用三步进入C++：

第一步：简单地把C++作为一个“ 更好的C ”，因为C++在使用任何函数之前必须声明它，并且对于如何使用变量有更苛刻的要求。简单地用C++编译器编译C程序常常会发现错误。

第二步：进入“ 面向对象 ”的C++。这意味着，很容易看到将数据结构和在它上面活动的函数捆绑在一起的代码组织，看到构造函数和析构函数的价值，也许还会看到一些简单的继承，这是有好处的。许多用过C的程序员很快就知道这是有用的，因为无论何时，创建库时，这些都是要做的。然而在C++中，由编译器来帮我们完成这些工作。

在基于对象层上，我们可能受骗，因为无须花费太多精力就能得到很多好处。它也很容易使我们感到正在创建数据类型——制造类和对象，向这些对象发送消息，一切漂亮优美。

但是，不要犯傻，如果我们停留在这里，我们就失去了这个语言的最重要的部分。这个最重要的部分才真正是向面向对象程序设计的飞跃。要做到这一点，只有靠第三步。

第三步：使用虚函数。虚函数加强类型概念，而不是只在结构内和墙后封装代码，所以毫无疑问，对于新C++程序员，它们是最困难的概念。然而，它们也是理解面向对象程序设计的转折点。如果不用虚函数，就等于还不懂得OOP。

因为虚函数是与类型概念紧密联系的，而类型是面向对象的程序设计的核心，所以在传统的过程语言中没有类似于虚函数的东西。作为一个过程程序员，没有以往的参考可以帮助他思考虚函数，因为接触的是这个语言的其他特征。过程语言中的特征可以在算法层上理解，而虚函数只能用设计的观点理解。

14.1　向上映射

在上一章中，我们已经看到对象如何作为它自己的类型或它的基类的对象使用。另外，它

还能通过基类的地址被操作。取一个对象的地址（或指针或引用），并看作基类的地址，这被称为向上映射，因为继承树是以基类为顶点的。

我们看到会出现一个问题，这表现在下面的代码中：

```
//: WIND2.CPP -- Inheritance & upcasting
#include <iostream.h>
enum note { middleC, Csharp, Cflat }; // Etc.

class instrument {
public:
  void play(note) const {
    cout << "instrument::play" << endl;
  }
};

// Wind objects are instruments
// because they have the same interface:
class wind : public instrument {
public:
  // Redefine interface function:
  void play(note) const {
    cout << "wind::play" << endl;
  }
};

void tune(instrument& i) {
  // ...
  i.play(middleC);
}

main() {
  wind flute;
  tune(flute); // Upcasting
}
```

函数tune()（通过引用）接受一个instrument，但也不拒绝任何从instrument派生的类。在main()中，可以看到，无需映射，就能将wind对象传给tune()。这是可接受的，在instrument 中的接口必然存在于wind中，因为wind是公共的从instrument继承而来的。wind到instrument的向上映射会使wind的接口“变窄”，但它不能变得比instrument的整个接口还小。

这对于处理指针的情况也是正确的，唯一的不同是用户必须显式地取对象的地址，传给这个函数。

14.2 问题

WIND2.CPP的问题可以通过运行这个程序看到，输出是 instrument::play。显然，这不是所

希望的输出，因为我们知道这个对象实际上是 wind而不只是一个instrument。这个调用应当输出wind::play。为此，由instrument派生的任何对象应当使它的play版本被使用。

然而，当对函数用C方法时，WIND2.CPP的表现并不奇怪。为了理解这个问题，需要知道捆绑的概念。

函数调用捆绑

把函数体与函数调用相联系称为捆绑（binding）。当捆绑在程序运行之前（由编译器和连接器）完成时，称为早捆绑。我们可能没有听到过这个术语，因为在过程语言中是不会有的：C编译只有一种函数调用，就是早捆绑。

上面程序中的问题是早捆绑引起的，因为编译器在只有 instrument地址时它不知道正确的调用函数。

解决方法被称为晚捆绑，这意味着捆绑在运行时发生，基于对象的类型。晚捆绑又称为动态捆绑或运行时捆绑。当一个语言实现晚捆绑时，必须有一种机制在运行时确定对象的类型和合适的调用函数。这就是，编译器还不知道实际的对象类型，但它插入能找到和调用正确函数体的代码。晚捆绑机制因语言而异，但可以想象，一些种类的类型信息必须装在对象自身中。稍后将会看到它是如何工作的。

14.3 虚函数

对于特定的函数，为了引起晚捆绑，C++要求在基类中声明这个函数时使用 virtual关键字。晚捆绑只对 virtual起作用，而且只发生在我们使用一个基类的地址时，并且这个基类中有virtual函数，尽管它们也可以在更早的基类中定义。

为了创建一个virtual成员函数，可以简单地在这个函数声明的前面加上关键字 virtual。对于这个函数的定义不要重复，在任何派生类函数重定义中都不要重复它（虽然这样做无害）。如果一个函数在基类中被声明为 virtual，那么在所有的派生类中它都是 virtual的。在派生类中virtual函数的重定义通常称为越位。

为了从WIND2.CPP中得到所希望的结果，只需简单地在基类中的 play()之前增加virtual关键字：

```
//: WIND3.CPP -- Late binding with virtual
#include <iostream.h>
enum note { middleC, Csharp, Cflat }; // Etc.

class instrument {
public:
  virtual void play(note) const {
    cout << "instrument::play" << endl;
  }
};

// Wind objects are instruments
// because they have the same interface:
class wind : public instrument {
public:
```

```
  // Redefine interface function:
  void play(note) const {
    cout << "wind::play" << endl;
  }
};

void tune(instrument& i) {
  // ...
  i.play(middleC);
}

main() {
  wind flute;
  tune(flute); // Upcasting
}
```

这个文件除了增加了virtual关键字之外，一切与WIND2.CPP相同，但结果明显不一样。现在的输出是wind::play。

扩展性

通过将play( )在基类中定义为virtual，不用改变tune()函数就可以在系统中随意增加新函数。在一个设计好的 OOP程序中，大多数或所有的函数都沿用 tune()模型，只与基类接口通信。这样的程序是可扩展的，因为可以通过从公共基类继承新数据类型而增加新功能。操作基类接口的函数完全不需要改变就可以适合于这些新类。

现在，instrument例子有更多的虚函数和一些新类，它们都能与老的版本一起正确工作，不用改变tune()函数：

```
//: WIND4.CPP -- Extensibility in OOP
#include <iostream.h>
enum note { middleC, Csharp, Cflat }; // Etc.

class instrument {
public:
  virtual void play(note) const {
    cout << "instrument::play" << endl;
  }
  virtual char* what() const {
    return "instrument";
  }
  // Assume this will modify the object:
  virtual void adjust(int) {}
};

class wind : public instrument {
public:
  void play(note) const {
```

```
    cout << "wind::play" << endl;
  }
  char* what() const { return "wind"; }
  void adjust(int) {}
};

class percussion : public instrument {
public:
  void play(note) const {
    cout << "percussion::play" << endl;
  }
  char* what() const { return "percussion"; }
  void adjust(int) {}
};
class string : public instrument {
public:
  void play(note) const {
    cout << "string::play" << endl;
  }
  char* what() const { return "string"; }
  void adjust(int) {}
};

class brass : public wind {
public:
  void play(note) const {
    cout << "brass::play" << endl;
  }
  char* what() const { return "brass"; }
};

class woodwind : public wind {
public:
  void play(note) const {
    cout << "woodwind::play" << endl;
  }
  char* what() const { return "woodwind"; }
};

// Identical function from before:
void tune(instrument& i) {
  // ...
  i.play(middleC);
}

// New function:
```

```
void f(instrument& i) { i.adjust(1); }

// Upcasting during array initialization:
instrument* A[] = {
  new wind,
  new percussion,
  new string,
  new brass
};

main() {
  wind flute;
  percussion drum;
  string violin;
  brass flugelhorn;
  woodwind recorder;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);
  f(flugelhorn);
}
```

可以看到，这个例子已在wind之下增加了另外的继承层，但virtual机制正确工作，不管这里有多少层。adjust()函数不对于brass和woodwind重定义。当出现这种情况时，自动使用先前的定义——编译器保证虚函数总是有定义的，所以，决不会最终出现调用不与函数体捆绑的情况。（这种情况将意味着灾难。）

数组A[]存放指向基类instrument 的指针，所以在数组初始化过程中发生向上映射。这个数组和函数f()将在稍后的讨论中用到。

在对tune()的调用中，向上映射在对象的每一个不同的类型上完成。期望的结果总是能得到。这可以被描述为“发送消息给一对象和让这个对象考虑用它来做什么”。virtual函数是在试图分析项目时使用的透镜：基类应当出现在哪里？应当如何扩展这个程序？然而，在程序最初创建时，即便我们没有发现合适的基类接口和虚函数，在稍后甚至更晚，当我们决定扩展或维护这个程序时，我们也常常会发现它们。这不是分析或设计错误，它只意味着一开始我们还没有所有的信息。由于C++严格的模块化，这并不是大问题。因为当我们对系统的一部分作了修改时，往往不会象C那样波及系统的其他部分。

14.4 C++如何实现晚捆绑

晚捆绑如何发生？所有的工作都由编译器在幕后完成。当我们告诉它去晚捆绑时（用创建虚函数告诉它），编译器安装必要的晚捆绑机制。因为程序员常常从理解 C++虚函数机制中受益，所以这一节将详细阐述编译器实现这一机制的方法。

关键字virtual告诉编译器它不应当完成早捆绑，相反，它应当自动安装实现晚捆绑所必须的所有机制。这意味着，如果我们对brass对象通过基类instrument地址调用play()，我们将得到

恰当的函数。

为了完成这件事，编译器对每个包含虚函数的类创建一个表（称为 VTABLE）。在VTABLE中，编译器放置特定类的虚函数地址。在每个带有虚函数的类中，编译器秘密地置一指针，称为vpointer（缩写为VPTR），指向这个对象的VTABLE。通过基类指针做虚函数调用时（也就是做多态调用时），编译器静态地插入取得这个VPTR，并在VTABLE表中查找函数地址的代码，这样就能调用正确的函数使晚捆绑发生。

为每个类设置VTABLE、初始化VPTR、为虚函数调用插入代码，所有这些都是自动发生的，所以我们不必担心这些。利用虚函数，这个对象的合适的函数就能被调用，哪怕在编译器还不知道这个对象的特定类型的情况下。

下面几节将对此做更详细地阐述。

14.4.1 存放类型信息

可以看到，在任何类中，不存在显式的类型信息。而先前的例子和简单的逻辑告诉我们，必须有一些类型信息放在对象中，否则，类型不能在运行时建立。实际上，类型信息被隐藏了。为了看到它，这里有一个例子，可以测试使用虚函数的类的长度，并与没有虚函数的类比较。

```
//: SIZES.CPP -- Object sizes vs. virtual funcs
#include <iostream.h>

class no_virtual {
  int a;
public:
  void x() const {}
  int i() const { return 1; }
};

class one_virtual {
  int a;
public:
  virtual void x() const {}
  int i() const { return 1; }
};
class two_virtuals {
  int a;
public:
  virtual void x() const {}
  virtual int i() const { return 1; }
};

main() {
  cout << "int: " << sizeof(int) << endl;
  cout << "no_virtual: "
       << sizeof(no_virtual) << endl;
  cout << "void* : " << sizeof(void*) << endl;
  cout << "one_virtual: "
```

```
        << sizeof(one_virtual) << endl;
    cout << "two_virtuals: "
        << sizeof(two_virtuals) << endl;
}
```

不带虚函数，对象的长度恰好就是所期望的：单个 int的长度。而带有单个虚函数的one_virtual，对象的长度是no_virtual的长度加上一个void指针的长度。它反映出，如果有一个或多个虚函数，编译器在这个结构中插入一个指针（VPTR）。在one_virtual 和 two_virtuals之间没有区别。这是因为VPTR指向一个存放地址的表，只需要一个指针，因为所有虚函数地址都包含在这个表中。

这个例子至少要求一个数据成员。如果没有数据成员， C++编译器会强制这个对象是非零长度，因为每个对象必须有一个互相区别的地址。如果我们想象在一个零长度对象的数组中索引，我们就能理解这一点。一个“哑”成员被插入到对象中，否则这个对象就有零长度。当virtual关键字插入类型信息时，这个“哑”成员的位置就被占用。在上面例子中，用注释符号将所有类的int a去掉，我们就会看到这种情况。

14.4.2 对虚函数作图

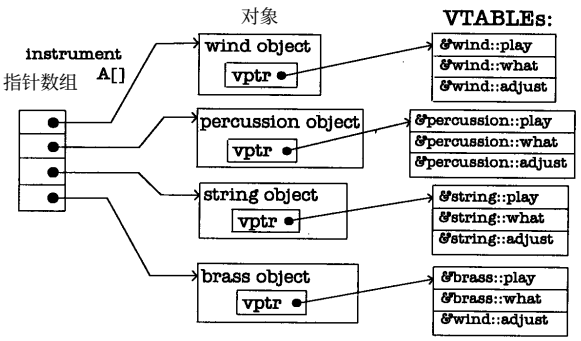


图 14-1

为了准确地理解使用虚函数时编译器做了些什么，使屏风之后进行的活动看得见是有帮助的。这里画的是在 14.3节WIND4.CPP中的指针数组A[]。

这个instrument指针数组没有特殊类型信息，它的每一个元素指向一个类型为instrument 的对象。wind、percussion、string和brass都适合这个范围，因为它们都是从instrument派生来的（并且和instrument有相同的接口和响应相同的消息），因此，它们的地址也自然能放进这个数组里。然而，编译器并不知道它们比instrument对象更多的东西，所以，留给它们自己处理，而通常调用所有函数的基类版本。但在这里，所有这些函数都被用 virtual声明，所以出现了不同的情况。每当创建一个包含有虚函数的类或从包含有虚函数的类派生一个类时，编译器就为这个类创建一个VTABLE，如这个图的右面所示。在这个表中，编译器放置了在这个类中或在它的基类中所有已声明为virtual的函数的地址。如果在这个派生类中没有对在基类中声明为 virtual的函数进行重新定义，编译器就使用基类的这个虚函数地址。（在brass的VTABLE中，adjust的入口就是这种情况。）然后编译器在这个类中放置 VPTR（可在SIZES.CPP中发现）。当使用简单继承时，对于每个对象只有一个VPTR。VPTR必须被初始化为指向相应的VTABLE。（这在构造函数中发生，在稍后会看得更清楚。）

一旦VPTR被初始化为指向相应的VTABLE，对象就“知道”它自己是什么类型。但只有当虚函数被调用时这种自我知识才有用。

通过基类地址调用一个虚函数时（这时编译器没有能完成早捆绑的足够的信息），要特殊处理。它不是实现典型的函数调用，对特定地址的简单的汇编语言 CALL，而是编译器为完成这个函数调用产生不同的代码。下面看到的是通过 instrument指针对于brass调用adjust()。

instrument引用产生如下结果：

编译器从这个instrument指针开始，这个指针指向这个对象的起始地址。所有的 instrument对象或由 instrument派生的对象都有它们的VPTR，它在对象的相同的位置（常常在对象的开头），所以编译器能够取出这个对象的VPTR。VPTR指向VTABLE的开始地址。所有的VTABLE有相同的顺序，不管何种类型的对象。play()是第一个，what()是第二个，adjust()是第三个。所以编译器知道adjust()函数必在VPTR+2处。这样，不是“以instrument::adjust地址调用这个函数”（这是早捆绑，是错误活动），而是产生代码，“在VPTR+2处调用这个函数”。因为VPTR的效果和实际函数地址的确定发生在运行时，所以这样就得到了所希望的晚捆绑。向这个对象发送消息，这个对象能断定它应当做什么。

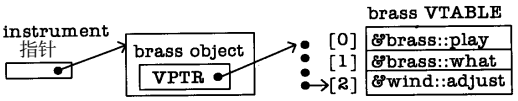


图 14-2

14.4.3 撩开面纱

如果能看到由虚函数调用而产生的汇编语言代码，这将是很有帮助的，这样可以看到后捆绑实际上是如何发生的。下面是在函数f（instrument&i）中调用

```
i.adjust(1);
```

某个编译器所产生的输出：

```
push   1
push   si
mov    bx,word ptr [si]
call   word ptr [bx+4]
add    sp,4
```

C++函数调用的参数与C函数调用一样，是从右向左进栈的（这个顺序是为了支持C的变量参数表），所以参数1首先压栈。在这个函数的这个地方，寄存器si（intel x86处理器的一部分）存放i的首地址。因为它是被选中的对象的首地址，它也被压进栈。记住，这个首地址对应于this的值，正因为调用每个成员函数时this都必须作为参数压进栈，所以成员函数知道它工作在哪个特殊对象上。这样，我们总能看到，在成员函数调用之前压栈的次数等于参数个数加一（除了static成员函数，它没有this）。

现在，必须实现实际的虚函数调用。首先，必须产生VPTR，使得能找到VTABLE。对于这个编译器，VPTR在对象的开头，所以this的内容对应于VPTR。下面这一行

```
mov bx，word ptr[si]
```

取出si（即this）所指的字，它就是VPTR。将这个VPTR放入寄存器bx中。

放在bx中的这个VPTR指向这个VTABLE的首地址，但被调用的函数在VTABLE中不是第0个位置，而是第二个位置（因为它是这个表中的第三个函数）。对于这种内存模式，每个函数指针是两个字节长，所以编译器对VPTR加四，计算相应的函数地址所在的地方，注意，这是编译时建立的常值。所以我们只要保证在第二个位置上的指针恰好指向adjust()。幸好编译器仔细处理，并保证在VTABLE中的所有函数指针都以相同的次序出现。

一旦在VTABLE中相应函数指针的地址被计算出来，就调用这个函数。所以取出这个地址并马上在这个句子中调用：

```
call word ptr [bx+4]
```

最后，栈指针移回去，以清除在调用之前压入栈的参数。在 C和C++汇编代码中，我们将常常看到调用者清除这些参数，但这依处理器和编译器的实现而有所变化。

14.4.4 安装vpointer

因为VPTR决定了对象的虚函数的行为，所以我们会看到 VPTR总是指向相应的VTABLE是多么重要。在VPTR适当初始化之前，我们绝对不能对虚函数调用。当然，能保证初始化的地点是在构造函数中，但是，在WIND例子中没有一个是有构造函数的。

这样，缺省构造函数的创建是很关键的。在 WIND例子中，编译器创建了一个缺省构造函数，它只做初始化 VPTR的工作。在能用任何 instrument对象做任何事情之前，对于任何instrument对象自动调用这个构造函数。所以，调用虚函数是安全的。

在构造函数中，自动初始化VPTR的含义在下一节讨论。

14.4.5 对象是不同的

认识到向上映射仅处理地址，这是重要的。如果编译器有一个它知道确切类型的对象，那么（在C++中）对任何函数的调用将不再用晚捆绑，或至少编译器不必须用晚捆绑。因为编译器知道对象的类型，为了提高效率，当调用这些对象的虚函数时，很多编译器使用早捆绑。下面是一个例子：

```
//: EARLY.CPP -- Early binding & virtuals
#include <iostream.h>

class base {
public:
  virtual int f() const { return 1; }
};

class derived : public base {
public:
  int f() const { return 2; }
};
main() {
  derived d;
  base* b1 = &d;
  base& b2 = d;
  base b3;
  // Late binding for both:
  cout << "b1->f() = " << b1->f() << endl;
  cout << "b2.f() = " << b2.f() << endl;
  // Early binding (probably):
  cout << "b3.f() = " << b3.f() << endl;
}
```

在b1->f()和b2.f()中，使用地址，就意味着信息不完全：b1和b2可能表示base的地址也可能表示其派生对象的地址，所以必须用虚函数。而当调用 b3.f()时不存在含糊，编译器知道确切的类型和知道它是一个对象，所以它不可能是由 base派生的对象，而确切的只是一个 base。这

样，可以用早捆绑。但是，如果不希望编译器的工作如此复杂，仍可以用晚捆绑，并且有相同的结果。

14.5 为什么需要虚函数

在这个问题上，我们可能会问：“如果这个技术如此重要，并且能使得任何时候都能调用‘正确’的函数。那么为什么它是可选的呢？为什么我还需要知道它呢？”

问得好。回答关系到C++的基本哲学：“因为它不是相当高效率的”。从前面的汇编语言输出可以看出，它并不是对于绝对地址的一个简单的CALL，而是为设置虚函数调用需要多于两条复杂的汇编指令。这既需要代码空间，又需要执行时间。一些面向对象的语言已经接受了这种概念，即晚捆绑对于面向对象程序设计是性质所决定的，所以应当总是出现，它应当是不可选的，而且用户不应当必须知道它。这是由创造语言时的设计决定，而这种特殊的方法对于许多语言是合适的[1]。C++来自C传统，效率是重要的。创造C完全是为了代替汇编语言以实现操作系统（从而改写操作系统——Unix——使得比它的先驱更轻便）。希望有C++的主要理由之一是让C程序员效率更高[2]。C程序员遇到C++时提出的第一个问题是“我将得到什么样的规模和速度效果？”如果回答是“除了函数调用时需要有一点额外的开销外，一切皆好”，那么许多人就会仍使用C，而不会改变到C++。另外，内联函数是不可能的，因为虚函数必须有地址放在VTABLE中。所以虚函数是可选的，而且该语言的缺省是非虚拟的，这是最快的配置。Stroustrup声明他的方针是“如果我们不用它，我们就不会为它花费”。

因此，virtual关键字可以改变程序的效率。然而，设计我们的类时，我们不应当为效率问题而担心。如果我们想使用多态，就在每处使用虚函数。当我们试图加速我们的代码时，我们只需寻找能让它非虚的函数（在其他方面通常有更大的好处）。

有些证据表明，进入C++的规模和速度改进是在C的规模和速度的10%之内，并且常常更接近。能够得到更小的规模和更高速度的原因是因为C++可以有比用C更快的方法设计程序，而且设计的程序更小。

14.6 抽象基类和纯虚函数

在所有的instrument的例子中，基类instrument中的函数总是“假”函数。如果调用这些函数，就会指出已经做错了什么事。这是因为，instrument的目的是对所有从它派生来的类创建公共接口，如在下面的图中看到的：

虚线表示类（一个类只是一个描述，而不是一个物理实体——虚线代表了它的非物理的“性质”）。从派生类到基类的箭头表示继承关系。

建立公共接口的唯一的理由是使得它能对于每个不同的子类有不同的表示。它建立一个基本的格式，由此可以知道什么是对于所有派生类公共的。注意，另外一种表达方法是称instrument为抽象基类（或简称为抽象类），当希望通过公共接口

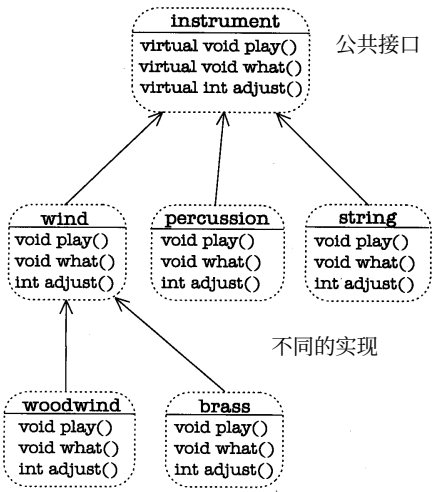


图 14-3

[1] 例如，Smalltalk用这种方法获得了很大的成功。

[2] 发明C++的贝尔实验室就是利用高效率为公司节约了大笔的费用。

操作一组类时就创建抽象类。

注意，只需在基类中声明函数为 virtual。与这个基类声明相匹配的所有派生类函数都将按照虚机制调用。当然我们也可以在派生类声明中使用 virtual 关键字（有些人为了清楚而这样做），但这是多余的。

如果我们有一个真实的抽象类（就像 instrument），这个类的对象几乎总是没有意义的。也就是说，instrument的含义只表示接口，不表示特例实现。所以创建一个 instrument对象没有意义。我们也许想防止用户这样做。这能通过让 instrument的所有虚函数打印出错信息而完成，但这种方法到运行时才能获得出错信息，并且要求用户可靠而详尽地测试。所以最好是在编译时就能发现这个问题。

C++对此提供了一种机制，称为纯虚函数。下面是它的声明语法：

```
virtual void x() = 0;
```

这样做，等于告诉编译器在VTABLE中为函数保留一个间隔，但在这个特定间隔中不放地址。只要有一个函数在类中被声明为纯虚函数，则 VTABLE就是不完全的。包含有纯虚函数的类称为纯抽象基类。

如果一个类的VTABLE是不完全的，当某人试图创建这个类的对象时，编译器做什么呢？由于它不能安全地创建一个纯抽象类的对象，所以如果我们试图制造一个纯抽象类的对象，编译器就发出一个出错信息。这样，编译器就保证了抽象类的纯洁性，我们就不用担心误用它了。

这是修改后的WIND4.CPP 14.3节，它使用了纯虚函数：

```
//: WIND5.CPP -- Pure abstract base classes
#include <iostream.h>
enum note { middleC, Csharp, Cflat }; // Etc.

class instrument {
public:
  // Pure virtual functions:
  virtual void play(note) const = 0;
  virtual char* what() const = 0;
  // Assume this will modify the object:
  virtual void adjust(int) = 0;
};
// Rest of the file is the same ...

class wind : public instrument {
public:
  void play(note) const {
    cout << "wind::play" << endl;
  }
  char* what() const { return "wind"; }
  void adjust(int) {}
};

class percussion : public instrument {
```

```
public:
  void play(note) const {
    cout << "percussion::play" << endl;
  }
  char* what() const { return "percussion"; }
  void adjust(int) {}
};

class string : public instrument {
public:
  void play(note) const {
    cout << "string::play" << endl;
  }
  char* what() const { return "string"; }
  void adjust(int) {}
};

class brass : public wind {
public:
  void play(note) const {
    cout << "brass::play" << endl;
  }
  char* what() const { return "brass"; }
};

class woodwind : public wind {
public:
  void play(note) const {
    cout << "woodwind::play" << endl;
  }
  char* what() const { return "woodwind"; }
};

// Identical function from before:
void tune(instrument& i) {
  // ...
  i.play(middleC);
}

// New function:
void f(instrument& i) { i.adjust(1); }

main() {
  wind flute;
  percussion drum;
  string violin;
```

```
  brass flugelhorn;
  woodwind recorder;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);
  f(flugelhorn);
}
```

纯虚函数是非常有用的，因为它们使得类有明显的抽象性，并告诉用户和编译器希望如何使用。

注意，纯虚函数防止对纯抽象类的函数以传值方式调用。这样，它也是防止对象意外使用值向上映射的一种方法。这样就能保证在向上映射期间总是使用指针或引用。

纯虚函数防止产生VTABLE，但这并不意味着我们不希望对其他函数产生函数体。我们常常希望调用一个函数的基类版本，即便它是虚拟的。把公共代码放在尽可能靠近我们的类层次根的地方，这是很好的想法。这不仅节省了代码空间，而且能允许使改变的传播变得容易。

纯虚定义

在基类中，对纯虚函数提供定义是可能的。我们仍然告诉编译器不要允许纯抽象基类的对象，而且纯虚函数在派生类中必须定义，以便于创建对象。然而，我们可能希望一块代码对于一些或所有派生类定义能共同使用，不希望在每个函数中重复这段代码，如下所示：

```
//: PVDEF.CPP -- Pure virtual base definition
#include <iostream.h>

class base {
public:
  virtual void v() const = 0;
  // In situ:
  virtual void f() const = 0 {
    cout << "base::f()\n";
  }
};

void base::v() const { cout << "base::v()\n";}

class d : public base {
public:
  // Use the common base code:
  void v() const { base::v(); }
  void f() const { base::f(); }
};

main() {
```

```
  d D;
  D.v();
  D.f();
}
```

在base VTABLE中的间隔仍然空着，但在这个派生类中刚好有一个函数，可以通过名字调用它。

这个特征的另外的好处是，它允许使用一个纯虚函数而不打乱已存在的代码。（这是一个处理没有重定义虚函数类的方法。）

14.7 继承和VTABLE

可以想象，当实现继承和定义一些虚函数时，会发生什么事情？编译器对新类创建一个新VTABLE表，并且插入新函数的地址，对于没有重定义的虚函数使用基类函数的地址。无论如何，在VTABLE中总有全体函数的地址，所以绝对不会对不在其中的地址调用。（否则损失惨重。）

但当在派生类中增加新的虚函数时会发生什么呢？这里有一个例子：

```
//: ADDV.CPP -- Adding virtuals in derivation
#include <iostream.h>

class base {
  int i;
public:
  base(int I) : i(I) {}
  virtual int value() const { return i; }
};
class derived : public base {
public:
  derived(int I) : base(I) {}
  int value() const {
    return base::value() * 2;
  }
  // New virtual function in the derived class:
  virtual int shift(int x) const {
    return base::value() << x;
  }
};

main() {
  base* B[] = { new base(7), new derived(7) };
  cout << "B[0]->value() = "
       << B[0]->value() << endl;
  cout << "B[1]->value() = "
       << B[1]->value() << endl;
//! cout << "B[1]->shift(3) = "
//!      << B[1]->shift(3) << endl; // Illegal
}
```

类base包含单个虚函数value()，而类derived增加了第二个称为shift()的虚函数，并重定义了value的含义。下图有助于显示发生的事情，其中有编译器为 base和derived创建的两个VTABLE。

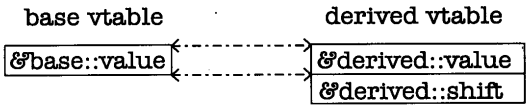


图 14-4

注意，编译器映射 derived VTABLE中的value地址位置等于在base VTABLE中的位置。类似的，如果一个类从derived继承而来，它的shift版本在它的VTABLE中的位置应当等于在derived中的位置。这是因为（正如通过汇编语言例子看到的）编译器产生的代码只是简单地在 VTABLE中用偏移选择虚函数。不论对象属于哪个特殊的类，它的VTABLE是以同样的方法设置的，所以对虚函数的调用将总是用同样的方法。

这样，编译器只对指向基类对象的指针工作。而这个基类只有 value函数，所以它就是编译器允许调用的唯一的函数。那么，如果只有指向基类对象的指针，那么编译器怎么可能知道自己正在对derived对象工作呢？这个指针可能指向其他一些没有shift函数的类。在VTABLE中，可能有，也可能没有一些其他函数的地址，但无论何种情况，对这个VTABLE地址做虚函数调用都不是我们想要的。所以编译器防止对只在派生类中存在的函数做虚函数调用，这是幸运的，合乎逻辑的。

有一些很少见的情况：可能我们知道指针实际上指向哪一种特殊子类的对象。这时如果想调用只存在于这个子类中的函数，则必须映射这个指针。下面的语句可以纠正由前面程序产生的错误：

```
((derived*)B[1])->shift(3)
```

在这里我们碰巧知道B[1]指向derived对象，但这种情况很少见。如果我们的程序确定我们必须知道所有对象的准确的类型，那么我们应当重新考虑它，因为我们可能在进行不正确的虚函数调用。然而对于有些情况如果知道保存在一般包容器中的所有对象的准确类型，会使我们的设计工作在最佳状态（或没有选择）。这就是运行时类型辨认问题（简称RTTI）。

运行时类型辨认是有关映射基类指针向下到派生类指针的问题。（“向上”和“向下”是相对典型类图而言的，典型类图以基类为顶点。）向上映射是自动发生的，不需强制，因为它是绝对安全的。向下映射是不安全的，因为这里没有关于实际类型的编译信息，所以必须准确地知道这个类实际上是什么类型。如果把它映射成错误的类型，就会出现麻烦。

第18章将描述C++提供运行时类型信息的方法。

• 对象切片

当多态地处理对象时，传地址与传值有明显的不同。所有在这里已经看到的例子和将会看到的例子都是传地址的，而不是传值的。这是因为地址都有相同的长度 [1]，传派生类型（它通常稍大一些）对象的地址和传基类（它通常小一点）对象的地址是相同的。如前面解释的，使用多态的目的是让对基类对象操作的代码也能操作派生类对象。

如果使用对象而不是使用地址或引用进行向上映射，发生的事情会使我们吃惊：这个对象被“切片”，直到所剩下来的是适合于目的的子对象。在下面例子中可以看到通过检查这个对象的长度切片剩下来的部分。

```
//: SLICE.CPP -- Object slicing
#include <iostream.h>
```

[1] 实际上，并不是所有机器上的指针都是同样大小的。但就本书讨论的范围而言，它们可被认为是同样大小的。

```
class base {
  int i;
public:
  base(int I = 0) : i(I) {}
  virtual int sum() const { return i; }
};

class derived : public base {
  int j;
public:
  derived(int I = 0, int J = 0)
    : base(I), j(J) {}
  int sum() const { return base::sum() + j; }
};

void call(base b) {
  cout << "sum = " << b.sum() << endl;
}

main() {
  base b(10);
  derived d(10, 47);
  call(b);
  call(d);
}
```

函数call()通过传值传递一个类型为base的对象。然后对于这个base对象调用虚函数sum()。我们可能希望第一次调用产生 10，第二次调用产生57。实际上，两次都产生10。

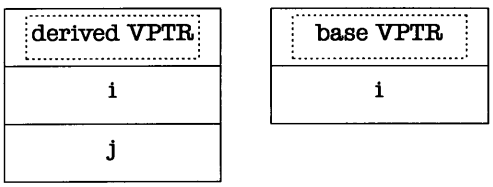


图 14-5

在这个程序中，有两件事情发生了。第一，call()接受的只是一个base对象，所以所有在这个函数体内的代码都将只操作与 base相关的数。对call()的任何调用都将引起一个与 base大小相同的对象压栈并在调用后清除。这意味着，如果一个由base派生来类对象被传给call，编译器接受它，但只拷贝这个对象对应于base的部分，切除这个对象的派生部分，如图：

现在，我们可能对这个虚函数调用感到奇怪：这里，这个虚函数既使用了base（它仍存在），又使用了derived的部分（derived不再存在了，因为它被切片）。

其实我们已经从灾难中被解救出来，这个对象正安全地以值传递。因为这时编译器认为它知道这个对象的确切的类型（这个对象的额外特征有用的任何信息都已经失去）。另外，用值传递时，它对base对象使用拷贝构造函数，该构造函数初始化 VPTR指向base VTABLE，并且只拷贝这个对象的base部分。这里没有显式的拷贝构造函数，所以编译器自动地为我们合成一个。由于上述诸原因，这个对象在切片期间变成了一个base对象。

对象切片实际上是去掉了对象的一部分，而不是象使用指针或引用那样简单地改变地址的

内容。因此，对象向上映射不常做，事实上，通常要提防或防止这种操作。我们可以通过在基类中放置纯虚函数来防止对象切片。这时如果进行对象切片就将引起编译时的出错信息。

14.8 虚函数和构造函数

当创建一个包含有虚函数的对象时，必须初始化它的 VPTR以指向相应的VTABLE。这必须在有关虚函数的任何调用之前完成。正如我们可能猜到的，因为构造函数有使对象成为存在物的工作，所以它也有设置 VPTR的工作。编译器在构造函数的开头部分秘密地插入能初始化VPTR的代码。事实上，即使我们没有对一个类创建构造函数，编译器也会为我们创建一个带有相应VPTR初始化代码的构造函数（如果有虚函数）。这有几个含意。

首先这涉及效率。内联（inline）函数的理由是对小函数减少调用代价。如果C++不提供内联（inline）函数，预处理器就可能被用以创建这些“宏”。然而，预处理器没有通道或类的概念，因此不能被用以创建成员函数宏。另外，有了由编译器插入隐藏代码的构造函数，预处理宏根本不能工作。

当寻找效率漏洞时，我们必须明白，编译器正在插入隐藏代码到我们的构造函数中。这些隐藏代码不仅必须初始化 VPTR，而且还必须检查this的值（万一operator new返回零）和调用基类构造函数。放在一起，这些代码能影响我们认为是一个小内联函数的调用。特别是，构造函数的规模会抵消减少函数调用节省的费用。如果做大量的内联构造函数调用，我们的代码长度就会增长，而在速度上没有任何好处。

当然，我们也许并不会立即把这些小构造函数都变成非内联，因为它们更容易做为内联的来写。但是，当我们正在调整我们的代码时，记住，务必去掉这些构造函数的内联性。

14.8.1 构造函数调用次序

构造函数和虚函数的第二个有趣的方面涉及构造函数的调用顺序和在构造函数中虚函数调用的方法。

所有基类构造函数总是在继承类构造函数中被调用。这是有意义的，因为构造函数有一项专门的工作：确保对象被正确的建立。派生类只访问它自己的成员，而不访问基类的成员，只有基类构造函数能恰当地初始化它自己的成员。因此，确保所有的构造函数被调用是很关键的，否则整个对象不会适当地被构造。这就是为什么编译器强制构造函数对派生类的每个部分调用。如果不在构造函数初始化表达式表中显式地调用基类构造函数，它就调用缺省构造函数。如果没有缺省构造函数，编译器将报告出错。（在这个例子中， class x没有构造函数，所以编译器能自动创建一个缺省构造函数。）

构造函数调用的顺序是重要的。当继承时，我们必须完全知道基类和能访问基类的任何public 和protected成员。这也就是说，当我们在派生类中时，必须能肯定基类的所有成员都是有效的。在通常的成员函数中，构造已经发生，所以这个对象的所有部分的成员都已经建立。然而，在构造函数内，必须想办法保证所有我们的成员都已经建立。保证它的唯一方法是让基类构造函数首先被调用。这样，当我们在派生类构造函数中时，在基类中我们能访问的所有成员都已经被初始化了。在构造函数中，“必须知道所有成员对象是有效的”也是下面做法的理由：只要可能，我们应当在这个构造函数初始化表达式表中初始化所有的成员对象（放在合成的类中的对象）。只要我们遵从这个做法，我们就能保证所有基类成员和当前对象的成员对象已经被初始化。

14.8.2 虚函数在构造函数中的行为

构造函数调用层次会导致一个有趣的两难选择。试想；如果我们正在构造函数中并且调用虚函数，那么会发生什么现象呢？对于普通的成员函数，虚函数的调用是在运行时决定的，这是因为编译时并不能知道这个对象是属于这个成员函数所在的那个类，还是属于由它派生出来的类。于是，我们也许会认为在构造函数中也会发生同样的事情。

然而，情况并非如此。对于在构造函数中调用一个虚函数的情况，被调用的只是这个函数的本地版本。也就是说，虚机制在构造函数中不工作。

这个行为有两个理由。在概念上，构造函数的工作是把对象变成存在物。在任何构造函数中，对象可能只是部分被形成——我们只能知道基类已被初始化了，但不知道哪个类是从这个基类继承来的。然而，虚函数是“向前”和“向外”进行调用。它能调用在派生类中的函数。如果我们在构造函数中也这样做，那么我们所调用的函数可能操作还没有被初始化的成员，这将导致灾难的发生。

第二个理由是机械的。当一个构造函数被调用时，它做的首要的事情之一是初始化它的VPTR。因此，它只能知道它是“当前”类的，而完全忽视这个对象后面是否还有继承者。当编译器为这个构造函数产生代码时，它是为这个类的构造函数产生代码 --既不是为基类，也不是为它的派生类（因为类不知道谁继承它）。所以它使用的VPTR必须是对于这个类的VTABLE。而且，只要它是最后的构造函数调用，那么在这个对象的生命期内， VPTR将保持被初始化为指向这个VTABLE。但如果接着还有一个更晚派生的构造函数被调用，这个构造函数又将设置VPTR指向它的VTABLE，等等，直到最后的构造函数结束。VPTR的状态是由被最后调用的构造函数确定的。这就是为什么构造函数调用是从基类到更加派生类顺序的另一个理由。

但是，当这一系列构造函数调用正发生时，每个构造函数都已经设置 VPTR指向它自己的VTABLE。如果函数调用使用虚机制，它将只产生通过它自己的 VTABLE的调用，而不是最后的VTABLE（所有构造函数被调用后才会有最后的 VTABLE）。另外，许多编译器认识到，如果在构造函数中进行虚函数调用，应该使用早捆绑，因为它们知道晚捆绑将只对本地函数产生调用。无论哪种情况，在构造函数中调用虚函数都没有结果。

14.9 析构函数和虚拟析构函数

构造函数不能是虚的（在附录B中的技术只类似于虚构造函数）。但析构函数能够且常常必须是虚的。

构造函数有其特殊的工作。它首先调用基本构造函数，然后调用在继承顺序中的更晚派生的构造函数，如此一块一块地把对象拼起来。类似的，析构函数也有一个特殊的工作——它必须拆卸可能属于某类层次的对象。为了做这些工作，它必须按照构造函数调用相反的顺序，调用所有的析构函数。这就是，析构函数自最晚派生的类开始，并向上到基类。这是安全且合理的：当前的析构函数能知道基类成员仍是有效的，因为它知道它是从哪一个派生而来的，但不知道从它派生出哪些。

应当记住，构造函数和析构函数是必须遵守调用层次唯一的地方。在所有其他函数中，只是某个函数被调用，而无论它是虚的还是非虚的。同一个函数的基类版本在通常的函数中被调用（无论虚否）的唯一的方法是直接地调用这个函数。

通常，析构函数的活动是很正常的。但是，如果我们想通过指向某个对象的基类的指针操纵这个对象（这就是，通过它的一般接口操纵这个对象），会发生什么现象呢？这在面向对象

的程序设计中确实很重要。当我们想delete在栈中已经用new创建的类的对象的指针时，就会出现这个问题。如果这个指针是指向基类的，编译器只能知道在 delete期间调用这个析构函数的基类版本。我们已经知道，虚函数被创建恰恰是为了解决同样的问题。幸好，析构函数可以是虚函数，于是一切问题就迎刃而解了。

虽然析构函数象构造函数一样，是“ 例外 ”函数，但析构函数可以是虚的，这是因为这个对象已经知道它是什么类型（而在构造期间则不然）。一旦对象已被构造，它的VPTR就已被初始化了，所以虚函数调用能发生。

如果我们创建一个纯虚析构函数，我们就必须提供函数体，因为（不像普通函数）在类层次中所有析构函数都总是被调用。这样，这个纯虚析构函数的函数体以调用结束。下面是例子：

```
//: PVDEST.CPP -- Pure virtual destructors
// require a function body.
#include <iostream.h>

class base {
public:
  virtual ~base() = 0 {
    cout << "~base()" << endl;
  }
};

class derived : public base {
public:
  ~derived() {
    cout << "~derived()" << endl;
  }
};

main() {
  base* bp = new derived; // Upcast
  delete bp; // Virtual destructor call
}
```

尽管在基类中的析构函数的纯虚性有强制继承者重定义这个析构函数的作用，但这个基类体仍作为析构函数的一部分被调用。

作为准则，任何时候在类中有虚函数，我们就应当直接增加虚析构函数（即便它什么事也不做）。这样，能保证以后不发生意外。

在析构函数中的虚机制

在析构期间，有一些我们可能不希望马上发生的情况。如果我们正在一个普通的成员函数中，并且调用一个虚函数，则这个函数被使用晚捆绑机制调用。而对于析构函数，这样不行，不论是虚的还是非虚的。在析构函数中，只有成员函数的本地版本被调用，虚机制被忽略。

为什么是这样呢？假设虚机制在析构函数中使用，那么调用下面这样的虚函数是可能的：这个函数是在继承层次中比当前的析构函数“ 更靠外 ”（更晚派生的）。但是，有一点我们要注意，析构函数从“ 外层 ”被调用（从最晚派生析构函数向基本析构函数）。所以，实际上被调

用的函数就可能操作在已被删除的对象上。因此，编译器决定在编译时只调用这个函数的“本地”版本。注意，对于构造函数也是如此（这在前面已讲到）。但在构造函数的情况下，这样做是因为信息还不可用，在析构函数中，信息（也就是VPTR）虽存在，但不可靠。

14.10 小结

多态性在C++中用虚函数实现，它有不同的形式。在面向对象的程序设计中，我们有相同的表面（在基类中的公共接口）和使用这个表面的不同的形式：虚函数的不同版本。

在这一章中，我们已经看到，理解甚至创建一个多态的例子，不用数据抽象和继承是不可能的。多态是不能隔离看待的特性（例如像const和switch语句），它必须同抽象与继承一起工作，它是类关系的一个重要方面。人们常常被C++的其他非面向对象的特性所混淆，例如重载和缺省参数，它们有时被作为面向对象的特性描述。不要犯傻，如果它不是晚捆绑它就不是多态。

为了在我们的程序中有效的使用多态等面向对象的技术，我们不能只知道让我们的程序包含单个类的成员和消息，而且还应当知道类的共性和它们之间的关系。虽然这需要很大的努力，但这是值得的，因为我们将更快地开发程序和更好地组织代码，得到可扩充的程序和更容易维护的代码。

多态完善了这个语言的面向对象特性，但在C++中，有两个更重要的特性：模板（第15章）和异常处理（第17章）。这些特性使我们的程序设计能力有很大的提高，就像面向对象的其他特性：抽象数据类型、继承和多态一样。

14.11 练习

1. 创建一个非常简单的“shape”层次：基类称为shape，派生类称为circle、square和triangle。在基类中定义一个虚函数draw()，再在这些派生类中重定义它。创建指向我们在堆中创建的shape对象的指针数组（这样就形成了指针向上映射）。并且通过基类指针调用draw()，检验这个虚函数的行为。如果我们的调试器支持，就用单步执行这个例子。

2. 修改练习1，使得draw()是纯虚函数。尝试创建一个类型为shape的对象。尝试在构造函数内调用这个纯虚函数，结果如何。给出draw()的一个定义。

3. 写出一个小程序以显示在普通成员函数中调用虚函数和在构造函数中调用虚函数的不同。这个程序应当证明两种调用产生不同的结果。

4. 在EARLY.CPP中，我们如何能知道编译器是用早捆绑还是晚捆绑进行调用？根据我们自己的编译器来确定。

5. （中级）创建一个不带成员和构造函数而只有一个虚函数的基类class X，创建一个从X继承的类class Y，它没有显式的构造函数。产生汇编代码并检验它，以确定X的构造函数是否被创建和调用，如果是的，这些代码做什么？解释我们的发现。X没有缺省构造函数，但是为什么编译器不报告出错？

6. （中级）修改练习5，让每个构造函数调用一个虚函数。产生汇编代码。确定在每个构造函数内VPTR在何处被赋值。在构造函数内编译器使用虚函数机制吗？确定为什么这些函数的本地版本仍被调用。

7. （高级）参数为传值方式传递的对象的函数调用如果不用早捆绑，则虚调用可能会侵入不存在的部分。这可能吗？写一些代码强制虚调用，看是否会引起冲突。解释这个现象，检验当对象以传值方式传递时会发生什么现象。

8. （高级）通过我们的处理器的汇编语言信息或者其他技术，找出简单调用所需的时间数及虚函数调用的时间数，从而得出虚函数调用需要多用多少时间。

第15章　模板和包容器类

包容器类常用于创建面向对象程序的构造模块(building block),它使得程序内部代码更容易构造。

一个包容器类可描述为容纳其他对象的对象。可把它想像成允许向它存储对象，而以后可以从中取出这些对象的高速暂存存储器或智能存储块。

包容器类非常重要，曾被认为是早期的面向对象语言的基础。例如，在 Smalltalk中，程序员把语言设想为带有类库的程序翻译器，而类库的重要部分就是包容器类。所以 C++编译器的供应商很自然地会为用户提供包容器类库。

像许多早期的别的C++库一样，早期的包容器类库仿效 Smalltalk的基于对象的层次结构，该结构非常适合Smalltalk，但是该结构在C++的使用中却带来了一些不便，因此有必要寻求另外的方法。

包容器类是解决不同类型的代码重用问题的另一种方法。继承和组合为对象代码的重用提供一种方法，而C++的模板特性为源代码的重用提供一种方法。

虽然C++模板是通用的编程工具，但当它们被引入该语言时它们似乎不支持基于对象的包容器类层次结构。新近版本的包容器类库则完全由模板构造，程序员可以很容易地使用。

本章首先介绍包容器类和采用模板实现包容器类的方法，接着给出一些包容器类和怎样使用它们的例子。

15.1　包容器和循环子

假若打算用C语言创建一个堆栈，我们需要构造一个数据结构和一些相关函数，而在 C++中，则把二者封装在一个抽象数据类型内。下面的stack类是一个栈类的例子，为简化起见，它仅处理整数：

```
//: ISTACK.CPP -- Simple integer stack
#include <assert.h>
#include <iostream.h>

class istack {
  enum { ssize = 100 };
  int stack[ssize];
  int top;
public:
  istack() : top(0) { stack[top] = 0; }
  void push(int i) {
    if(top < ssize) stack[top++] = i;
  }
  int pop() {
    return stack[top > 0 ? --top : top];
  }
```

```
  friend class istackIter;
};

// An iterator is a "super-pointer":
class istackIter {
  istack& S;
  int index;
public:
  istackIter(istack& is)
    : S(is), index(0) {}
  int operator++() { // Prefix form
    if (index < S.top - 1) index++;
    return S.stack[index];
  }
  int operator++(int) { // Postfix form
    int returnval = S.stack[index];
    if (index < S.top - 1) index++;
    return returnval;
  }
};
// For interest, generate Fibonacci numbers:
int fibonacci(int N) {
  const sz = 100;
  assert(N < sz);
  static F[sz]; // Initialized to zero
  F[0] = F[1] = 1;
  // Scan for unfilled array elements:
  int i;
  for(i = 0; i < sz; i++)
    if(F[i] == 0) break;
  while(i <= N) {
    F[i] = F[i-1] + F[i-2];
    i++;
  }
  return F[N];
}

main() {
  istack is;
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  // Traverse with an iterator:
  istackIter it(is);
  for(int j = 0; j < 20; j++)
    cout << it++ << endl;
```

```
    for(int k = 0; k < 20; k++)
      cout << is.pop() << endl;
}
```

类istack是最为常见的自顶向下式的堆栈的例子。为了简化，此处栈的尺寸是固定的，但是也可以对其修改，通过把存储器安排在堆中分配内存，来自动地扩展其长度（后面的例子会介绍）。

第二个类istackIter是循环子的例子，我们可以把其当作仅能和 istack协同工作的超指针。注意，istackIter是istack的友元，它能访问istack的所有私有成员。

像一个指针一样， istackIter的工作是扫视istack并能在其中取值。在上述的简单的例子中，istackIter可以向前移动（利用运算符++的前缀和后缀形式）和取值。然而，此处却并没有对定义循环子方法予以限制。完全可以允许循环子在相关包容器中以任何方法移动和对包容的值进行修改。可是，按照惯例，循环子是由构造函数创建的，它只与一个包容器对象相连，并且在生命周期中不重新相连。（大多数循环子较小，所以我们可以容易地创建其他循环子。）

为了使例子更有趣，这个 fibonacci函数产生传统的“ 兔子繁殖数 ”，这是一个相当有效的实现，因为它决不会多次产生这些数。

在主程序main()中，我们可以看到栈和它的相关循环子的创建和使用。一旦创建了这些类，便可以很方便的使用它们。

包容器的必要性

很明显，一个整数堆栈不是一个重要的工具。包容器类的真正的需求是在堆上使用 new创建对象和使用delete析构对象的时候体现的。一个普遍的程序设计问题是程序员在写程序时不知道将创建多少对象。例如在设计航空交通控制系统时不应限制飞机的数目，不希望由于实际飞机的数目超过设计值而导致系统终止。在 CAD系统设计中，可以安排许多造型，但是只有用户能确定到底需要多少造型。我们一旦注意到上述问题，便可在程序开发中发现许多这样的例子。

依赖虚存储器去处理“ 存储器管理 ”的 C程序员常常发现 new、delete和包容器类思想的混乱。表面上看，创建一个囊括任何可能需求的 huge型全局数组是可行的，这不必有很多考虑（并不需要弄清楚 malloc()和free()）,但是这样的程序移植性较差，而且暗藏着难以捕捉的错误。

另外，创建一个huge型全局数组对象，构造函数和析构函数的开销会使系统效率显著地下降。C++中有更好的解决方法：将所需要的对象用 new创建并将其指针放入包容器中，待实际使用时将其取出。该方法确定了只有在绝对需要时才真正创建对象。所以在启动系统时可以忽略初始化条件，在环境相关的事件发生时才真正创建对象。

在大多数情况下，我们应当创建存放感兴趣的对象的包容器，应当用 new创建对象，然后把结果指针放在包容器中（在这个过程中向上映射），具体使用时再将指针从包容器中取出。该技术有很强的灵活性且易于分类组织。

15.2 模板综述

现在出现了一个问题， istack可存放整数，但是也应该允许存放造型、航班、工厂等等数据对象，如果这种改变每次都依赖源码的更新，则不是一个明智的办法。应该有更好的重用

方法。

有三种源代码重用方法：用于契约的 C方法；影响过C++的Smalltalk方法；C++的模板方法。

15.2.1 C方法

毫无疑问，应该摒弃C方法，这是由于它表现繁琐、易发生错误、缺乏美感。如果需要拷贝stack的源码并对其手工修改，还会带入新的错误。这是非常低效的技术。

15.2.2 Smalltalk 方法

Smalltalk方法是通过继承来实现代码重用的，既简单又直观。每个包容器类包含基本通用类object的所属项目。Smalltalk的基类库十分重要，它是创建类的基础。创建一个新类必须从已有类中继承。可以从类库中选择功能和需求接近的已有类作为父类，并在对父类的继承中加以修正从而创建一个新类。很明显这种方法可以减少我们的工作而提高效率（因此花大量的时间去学习Smalltalk类库是成为熟练的Smalltalk程序员的必由之路）。

所以这意味着Smalltalk的所有类都是单个继承树的一部份。当创建新类时必须继承树的某一枝。大多数树是已经存在的（它是Smalltalk的类库），树的根称作object——每个Smalltalk包容器所包含的相同的类。

这种方法表现出的整洁明了在于Smalltalk类层次上的任何类都源于object的派生，所以任何包容器可容纳任何类，包括包容器本身。基于基本通用类的（常称为 object）的单树形层次模式称为“ 基于对象的继承 ”。我们可能听说过这个概念，并猜想这是另一个 OOP的基本概念，就像“ 多态性 ”一样。但实际上这仅仅意味着以object（或相近的名称）为根的树形类结构和包含object的包容器类。

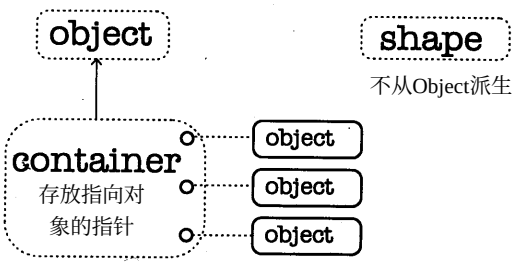


图 15-1

由于Smalltalk类库的发展史较C++更长久，且早期的 C++编译器没有包容器类库，所以C++能将Smalltalk类库的良好思想加以借鉴。这种借鉴出现在早期的C++实现中 [1]，由于它表现为一个有效的代码实体，许多人开始使用它，但把它用于包容器类的编程时则发现了一个问题。

该问题在于，在 Smalltalk中，我们可以强迫人们从单个层次结构中派生任何东西，但在C++中则不行。我们本来可能拥有完善的基于 object的层次结构以及它的包容器类，但是当我们从其他不用这种层次结构的供应商那里购买到一组类时，如造型类、航班类等等（层次结构增加开销，而C程序员可以避免这种情况），我们如何把这些类树集成进基于object的层次结构的包容器中呢？这些问题如下所示：

由于C++支持多个无关联层次结构，所以 Smalltalk的“ 基于object的层次结构 ”不能很好地工作。

解决方案似乎是明显的。如果我们有许多继承层次结构，我们就应当能从多个类继承：多重继承可以解决上述问题。所以我们应按下述的方法去实施：

[1] OOPS库，Keith Gorlen在NIH时开发的。一般以公开源代码的形式使用。

oshape具有shape的特点和行为，但它也是object的派生类，所以可将其置于包容器内。

但是原先的C++并不包含多重继承，当包容器问题出现时，C++供应商被迫去增加多重继承的特性。另外一些程序员一直认为多重继承不是一个好主意，因为它增加了不必要的复杂性。那时一句再三重复的话是“C++不是Smalltalk”，这意味着“不要把基于object的层次结构用于包容器类”。但最终[1]，由于不断的压力，还是把多重继承加入到该语言中了。编译器供应商将基于object的包容器类层次结构加入产品中并进行了调整，它们中的大多数由模板来替代。我们可以为多重继承是否可以解决大多数编程问题而进行争论，但是，在下一章中可以看到，由于其复杂性，除某些特殊情况，最好避免使用它。

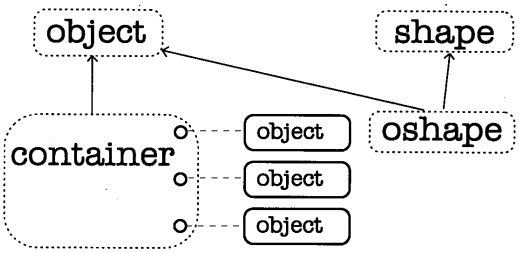


图 15-2

15.2.3 模板方法

尽管具有多重继承的基于对象的层次结构在概念上是直观的，但是在实践上较为困难。在Stroustrup的最初著作[2]中阐述了基于对象层次的一种更可取的选择。包容器类被创造作为参数化类型的大型预处理宏，而不是带自变量的模板，这些自变量能为我们所希望的类型替代。当我们打算创建一个包容器存放某个特别类型时，应当使用一对宏调用。

不幸的是，上述方法在当时的Smalltalk文献中被弄混淆了，加之难以处理，基本上没有什么人将其澄清。

在此期间，Stroustrup和贝尔实验室的C++小组对原先的宏方法进行了修正，对其进行了简化并将它从预处理范围移入了编译器。这种新的代码替换装置被称为模板[3]，而且它表现了完全不同的代码重用方法：模板对源代码进行重用，而不是通过继承和组合重用对象代码。包容器不再存放称为object的通用基类，而由一个非特化的参数来代替。当用户使用模板时，参数由编译器来替换，这非常像原来的宏方法，却更清晰、更容易使用。

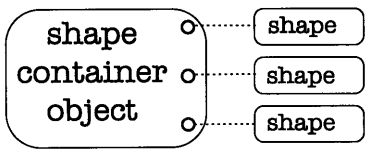


图 15-3

现在，使用包容器类时关于继承和组合的忧虑可以消除了，我们可以采用包容器的模板版本并且复制出和我们的问题相关的特定版本，像这样：

编译器会为我们做这些工作，而我们最终是以所需要的包容器去做我们的工作，而不是用那些令人头疼的继承层次。在C++中，模板实现了参数化类型的概念。模板方法的另一好处是对继承不熟悉、不适应的程序新手能正确地使用密封的包容器类。

15.3 模板的语法

“模板（template）”这一关键字会告诉编译器下面的类定义将操作一个或更多的非特定类型。当对象被定义时，这些类型必须被指定以使编译器能够替代它们。

---

[1] 我们也许决不能知道其全部，因为该语言的控制仍在AT&T中。

[2] The C++Programming Language，由Bjarne Stroustrup著（第一版，Addision-Wesley, 1986）。

[3] 模板的灵感最初出现在ADA。

下面是一个说明模板语法的小例子：

```
//: STEMP.CPP -- Simple template example
#include <iostream.h>
#include <assert.h>

template<class T>
class array {
  enum { size = 100 };
  T A[size];
public:
  T& operator[](int index) {
    assert(index >= 0 && index < size);
    return A[index];
  }
};
main() {
  array<int> ia;
  array<float> fa;
  for(int i = 0; i < 20; i++) {
    ia[i] = i * i;
    fa[i] = float(i) * 1.414;
  }
  for(int j = 0; j < 20; j++)
    cout << j << ": " << ia[j]
         << ", " << fa[j] << endl;
}
```

除了这一行

```
template<class T>
```

以外，它看上去像一个通常的类。这里 T是替换参数，它表示一个类型名称。在包容器类中，它将出现在那些原本由某一特定类型出现的地方。

在array中，其元素的插入和取出都用相同的函数，即重载的 operator[]来实现。它返回一个引用，因此可被用于等号的两边。注意，当下标值越界时，标准 C库的宏assert()将输出提示信息（使用assert()而不是allege()在于我们可以在调试后彻底移去测试代码）。这里，抛出一个异常，并由类的用户处理它会更适合一些，关于这些将在第 17章中做进一步介绍。

在main()中，我们可以看到非常容易地创建包含了不同类型对象的数组。当我们说：

```
array<int> ia;
array<float> fa;
```

这时，编译器两次扩展了数组模板（这被称为实例），创建两个新产生的类，我们可以把它们当作array_int和array_float（不同的编译器对名称有不同的修饰方法）。这些类就像手工创建的一样，除非你定义对象ia和fa，编译器会为你创建它们。注意要避免类在编译和连接中被重复定义。

15.3.1 非内联函数定义

当然，有时我们使用非内联成员函数定义，这时编译器会在成员函数定义之前察看模板声

明。下面在前述例子的基础上加以修正来说明非内联成员函数的定义：

```
//: STEMP2.CPP -- Non-inline template example
#include <assert.h>

template<class T>
class array {
  enum { size = 100 };
  T A[size];
public:
  T& operator[](int index);
};

template<class T>
T& array<T>::operator[](int index) {
  assert(index >= 0 && index < size);
  return A[index];
}

main() {
  array<float> fa;
  fa[0] = 1.414;
}
```

注意，在成员函数的定义中，类名称被限制为模板参数类型：array<T>。

你可以想象在一些混合型中编译器实际支持两个名字和参数类型。

• 头文件

甚至是在定义非内联函数时，模板的头文件中也会放置所有的声明和定义。这似乎违背了通常的头文件规则：“不要在分配存储空间前放置任何东西”，这条规则是为了防止在连接时的多重定义错误。但模板定义很特殊。由template<…>处理的任何东西都意味着编译器在当时不为它分配存储空间，它一直处于等待状态直到被一个模板实例告知。在编译器和连接器的某一处，有一机制能去掉指定模板的多重定义。所以为了容易使用，几乎总是在头文件中放置全部的模板声明和定义。

有时，也可能为了满足特殊的需要（例如，强制模板实例仅存在于一简单的 Windows DLL 文件中）而要在一个独立的CPP文件中放置模板的定义。大多数编译器有一些机制允许这么做，那么我们就必须检查我们特定的编译器说明文档以便使用它。

15.3.2 栈模板(the stack as a template)

对于ISTACK.CPP的包容器和循环子(第15.1节)，可以使用模板，作为普通包容器类实现：

```
//: STACKT.H -- Simple stack template
#ifndef STACKT_H_
#define STACKT_H_
template<class T> class stacktIter; // declare

template<class T>
class stackt {
```

```
  enum { ssize = 100 };
  T stack[ssize];
  int top;
public:
  stackt() : top(0) { stack[top] = 0; }
  void push(const T& i) {
    if(top < ssize) stack[top++] = i;
  }
  T pop() {
    return stack[top > 0 ? --top : top];
  }
  friend class stacktIter<T>;
};

template<class T>
class stacktIter {
  stackt<T>& S;
  int index;
public:
  stacktIter(stackt<T>& is)
    : S(is), index(0) {}
  T& operator++() { // Prefix form
    if (index < S.top - 1) index++;
    return S.stack[index];
  }
  T& operator++(int) { // Postfix form
    int returnIndex = index;
    if (index < S.top - 1) index++;
    return S.stack[returnIndex];
  }
};
#endif // STACKT_H_
```

注意在引用模板名称的地方，必须伴有该模板的参数列表，如 stackt<T>& S。我们可以想象，模板参数表中的参数将被重组，以对于每一个模板实例产生唯一的类名称。

同时也注意到，模板会对它包含的对象做一定的假设。例如，在 push()函数中，stackt会认为T的内部有一种赋值运算。

这里有一个修正过的例子用于检验模板：

```
//: STACKT.CPP -- Test simple stack template
#include <assert.h>
#include <iostream.h>
#include "..\14\stackt.h"
// For interest, generate Fibonacci numbers:
int fibonacci(int N) {
  const sz = 100;
```

```
  assert(N < sz);
  static F[sz]; // Initialized to zero
  F[0] = F[1] = 1;
  // Scan for unfilled array elements:
  int i;
  for(i = 0; i < sz; i++)
    if(F[i] == 0) break;
  while(i <= N) {
    F[i] = F[i-1] + F[i-2];
    i++;
  }
  return F[N];
}

main() {
  stackt<int> is;
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  // Traverse with an iterator:
  stacktIter<int> it(is);
  for(int j = 0; j < 20; j++)
    cout << it++ << endl;
  for(int k = 0; k < 20; k++)
    cout << is.pop() << endl;
}
```

唯一的不同是在实例is和it的创建中：我们指明了栈和循环子应该存放在模板参数表内部对象的类型。

15.3.3 模板中的常量

模板参数并不局限于有类定义的类型，可以使用编译器内置类型。这些参数值在模板特定实例时变成编译期间常量。我们甚至可以对这些参数使用缺省值：

```
//: MBLOCK.CPP -- Built-in types in templates
#include <assert.h>
#include <iostream.h>

template<class T, int size = 100>
class mblock {
  T array[size];
public:
  T& operator[](int index) {
    assert(index >= 0 && index < size);
    return array[index];
  }
};
```

```
class number {
  float f;
public:
  number(float F = 0.0f) : f(F) {}
  number& operator=(const number& n) {
    f = n.f;
    return *this;
  }
  operator float() const { return f; }
  friend ostream&
    operator<<(ostream& os, const number& x) {
      return os << x.f;
  }
};

template<class T, int sz = 20>
class holder {
  mblock<T, sz>* np;
public:
  holder() : np(0) {}
  number& operator[](int i) {
    assert(i >= 0 && i < sz);
    if(!np) np = new mblock<T, sz>;
    return np->operator[](i);
  }
};

main() {
  holder<number, 20> H;
  for(int i = 0; i < 20; i++)
    H[i] = i;
  for(int j = 0; j < 20; j++)
    cout << H[j] << endl;
}
```

类mblock是一个可选的数组对象，我们不能在其边界以外进行索引。（如果出现这种情况，将在第17章中介绍比assert()更好的方法。）

类holder和mblock极为相似，但是在 holder中有一个指向 mblock的指针，而不是含有mblock类型的嵌入式对象。该指针并不在 holder的构造函数中初始化，其初始化过程被安排在第一次访问的时候。 如果我们正在创建大量的对象，却又不立即全部访问它们，可以用这种技术，以节省存储空间。

15.4 stash & stack模板

贯穿本书且不断修正更新的stash和stack都是真正的包容器类，所以将其转化成为模板是必要的。但是，首先需要解决一个有关包容器类的重要问题：当包容器释放一个指向对象的指针时，

会析构该对象吗？例如，当一个包容器对象失去了指针控制，它会析构所有它所指向的对象吗？

15.4.1 所有权问题

所有权问题是普遍关心的问题。对对象进行完全控制的包容器通常无需担心所有权问题，因为它清晰、直接、完全地拥有其包含的对象。但是若包容器内包含指向对象的指针（这种情况在C++中相当普遍，尤其在多态情况下），而这些指针很可能用于程序的其他地方，那么删除了该指针指向的对象会导致在程序的其他地方的指针对已销毁的对象进行引用。为了避免上述情况，在设计和使用包容器时，必须考虑所有权问题。

许多程序都非常简单，一个包容器所包含的指针所指向的对象都仅仅用于包容器本身。在这种情况下，所有权问题简单而直观：该包容器拥有这些指针所指向的对象。由于通常大多数都是上述情况，因此把包容器完全拥有包容器内的指针所指向的对象的情况定义为缺省情形。

处理所有权问题的最好方法是由用户程序员来选择。这常常用构造函数的一个参数来完成，它缺省地指明所有权（对于典型理想化的简单程序）。另外还有读取和设置函数用来查看和修正包容器的所有权。假若包容器内有删除对象的函数，包容器所有权的状态会影响删除，所以我们还可以找到在删除函数中控制析构的选项。我们可以对在包容器中的每一个成员添加所有权信息，这样，每个位置都知道它是否需要被销毁，这是一个引用记数变量，在这里是包容器而不是对象知道所指对象的引用数。

15.4.2 stash模板

“stash”类是一个理想的模板构造的实例,有关它的修改贯穿于本书(最近的见第12章)。下面的例子中，带有所有权操作的循环子也加入其中。

```
//: TSTASH.H -- PSTASH using templates
#ifndef TSTASH_H_
#define TSTASH_H_
#include <stdlib.h>
#include "..\allege.h"
// More convenient than nesting in tstash:
enum owns { no = 0, yes = 1, Default };
// Declaration required:
template<class Type, int sz> class tstashIter;

template<class Type, int chunksize = 20>
class tstash {
  int quantity;
  int next;
  owns own; // Flag
  void inflate(int increase = chunksize);
protected:
  Type** storage;
public:
  tstash(owns owns = yes);
  ~tstash();
  int Owns() const { return own; }
```

```
  void Owns(owns newOwns) { own = newOwns; }
  int add(Type* element);
  int remove(int index, owns d = Default);
  Type* operator[](int index);
  int count() const { return next; }
  friend class tstashIter<Type, chunksize>;
};

template<class Type, int sz = 20>
class tstashIter {
  tstash<Type, sz>& ts;
  int index;
public:
  tstashIter(tstash<Type, sz>& TS)
    : ts(TS), index(0) {}
  tstashIter(const tstashIter& rv)
    : ts(rv.ts), index(rv.index) {}
  // Jump interator forward or backward:
  void forward(int amount) {
    index += amount;
    if(index >= ts.next) index = ts.next -1;
  }
  void backward(int amount) {
    index -= amount;
    if(index < 0) index = 0;
  }
  // Return value of ++ and -- to be
  // used inside conditionals:
  int operator++() {
    if(++index >= ts.next) return 0;
    return 1;
  }
  int operator++(int) { return operator++(); }
  int operator--() {
    if(--index < 0) return 0;
    return 1;
  }
int operator--(int) { return operator--(); }
operator int() {
  return index >= 0 && index < ts.next;
}
Type* operator->() {
  Type* t = ts.storage[index];
  if(t) return t;
  allege(0,"tstashIter::operator->return 0");
```

```
    return 0; // To allow inlining
  }
  // Remove the current element:
  int remove(owns d = Default){
    return ts.remove(index, d);
  }
};

template<class Type, int sz>
tstash<Type, sz>::tstash(owns Owns) : own(Owns) {
  quantity = 0;
  storage = 0;
  next = 0;
}

// Destruction of contained objects:
template<class Type, int sz>
tstash<Type, sz>::~tstash() {
  if(!storage) return;
  if(own == yes)
    for(int i = 0; i < count(); i++)
      delete storage[i];
  free(storage);
}

template<class Type, int sz>
int tstash<Type, sz>::add(Type* element) {
  if(next >= quantity)
    inflate();
  storage[next++] = element;
  return(next - 1); // Index number
}

template<class Type, int sz>
int tstash<Type, sz>::remove(int index,owns d){
  if(index >= next || index < 0)
    return 0;
  switch(d) {
    case Default:
      if(own != yes) break;
    case yes:
      delete storage[index];
    case no:
      storage[index] = 0; // Position is empty
  }
  return 1;
```

```
}

template<class Type, int sz> inline
Type* tstash<Type, sz>::operator[](int index) {
  // No check in shipping application:
  assert(index >= 0 && index < next);
  return storage[index];
}

template<class Type, int sz>
void tstash<Type, sz>::inflate(int increase) {
  void* v =
    realloc(storage,
(quantity+increase)*sizeof(Type*));
  allegemem(v);  // Was it successful?
  storage = (Type**)v;
  quantity += increase;
}
#endif // TSTASH_H_
```

尽管枚举enum owns常常被嵌入在类中，但这里还是将其定义成全局量。这是更方便的使用方法，假若打算观察其效果，我们可以试着把它移进去。

storage指针在类中被置为保护方式，这样通过继承得到的类可以直接访问它。这意味着继承类必然依赖于tstash的特定实现，但是正如我们将在SORTED.CPP 例子(见15.7)中看到的，这样做是值得的。

own标志指明了包容器是否以缺省方式拥有它所包容的对象。如果是这样，存在于包容器中的指针所指向的对象将在析构函数中被相应地销毁。这是一种简单的方法，包容器知道它所包含的类型。可以在构造函数中用重载函数owns()读和修改缺省的所有权。

应该认识到，如果包容器存放的指针是指向基类的，该类型应该具备一个虚析构函数来保证正确地清除派生对象，置于包容器中的派生对象的地址已经被向上映射。

在生存期中tstashIter遵循着与单个包容器相结合的循环子模式。另外拷贝构造函数允许我们创建新的循环子，指向已存在的循环子所指向的位置，这样可以非常高效地在包容器中创建书签。forward() 和 backward()成员函数允许移动循环子几步，它和包容器边界有关。增量和减量的重载运算符可以移动循环子一个位置。循环子所涉及的元素常常用灵活的指针来对其操作，通过调用包容器中的remove()函数可完成对当前对象的消除。

下面的例子是创建和检测两个不同的 tstash对象，一个属于新类Int并在它的析构函数和构造函数中给出报告，而另一个含有属于第12章的String类的对象：

```
//: TSTEST.CPP -- Test TSTASH
#include <fstream.h>
#include "..\allege.h"
#include "..\14\tstash.h"
#include "..\11\strings.h"
const bufsize = 80;
ofstream out("tstest.out");
```

```
class Int {
  int i;
public:
  Int(int I = 0) : i(I) {
    out << ">" << i << endl;
  }
  ~Int() { out << "~" << i << endl; }
  operator int() const { return i; }
  friend ostream&
    operator<<(ostream& os, const Int& x) {
      return os << x.i;
  }
};

main() {
  tstash<Int> intStash; // Instantiate for int
  for(int i = 0; i < 30; i++)
    intStash.add(new Int(i));
  tstashIter<Int> Intit(intStash);
  Intit.forward(5);
  for(int j = 0; j < 20; j++, Intit++)
    Intit.remove(); // Default removal
  for(int k = 0; k < intStash.count(); k++)
    if(intStash[k]) // Remove() causes "holes"
      out << *intStash[k] << endl;

  ifstream file("tstest.cpp");
  allegefile(file);
  char buf[bufsize];
  // Instantiate for String:
tstash<String> stringStash;
while(file.getline(buf, bufsize))
  stringStash.add(makeString(buf));
for(int u = 0; u < stringStash.count(); u++)
  if(stringStash[u])
    out << *stringStash[u] << endl;
tstashIter<String> it(stringStash);
int j = 25;
it.forward(j);
while(it) {
  out << j++ << ": " << it->str() << endl;
  it++;
}
}
```

在两种情形中，都创建了循环子，用来在包容器中前后移动。请注意使用构造函数所产生

的优美效果：我们无需关心使用数组的实现细节。我们告诉包容器和循环子做什么，而不是怎么做，这将使得问题的解更容易形成概念，更容易建立和更容易修改。

15.4.3 stack模板

在第13章中的stack类，它既是一个包容器，也是一个带有相关循环子的模板。下面是新的头文件：

```
//: TSTACK.H -- Stack using templates
#ifndef TSTACK_H_
#define TSTACK_H_

// Declaration required:
template<class T> class tstackIterator;

template<class T>
class tstack {
  struct link {
    T* data;
    link* next;
    link(T* Data, link* Next) {
      data = Data;
      next = Next;
    }
  } * head;
  int owns;
public:
  tstack(int Owns = 1) : head(0), owns(Owns) {}
  ~tstack();
  void push(T* Data) {
    head = new link(Data,head);
  }
  T* peek() const { return head->data; }
  T* pop();
  int Owns() const { return owns; }
  void Owns(int newownership) {
    owns = newownership;
  }
  friend class tstackIterator<T>;
};

template<class T>
T* tstack<T>::pop() {
  if(head == 0) return 0;
  T* result = head->data;
  link* oldHead = head;
```

```
  head = head->next;
  delete oldHead;
  return result;
}

template<class T>
tstack<T>::~tstack() {
  link* cursor = head;
  while(head) {
    cursor = cursor->next;
    // Conditional cleanup of data:
    if(owns) delete head->data;
    delete head;
    head = cursor;
  }
}

template<class T>
class tstackIterator {
  tstack<T>::link* p;
public:
  tstackIterator(const tstack<T>& tl)
    : p(tl.head) {}
  tstackIterator(const tstackIterator& tl)
    : p(tl.p) {}
  // operator++ returns boolean indicating end:
  int operator++() {
    if(p->next)
      p = p->next;
    else p = 0; // Indicates end of list
    return int(p);
  }
  int operator++(int) { return operator++(); }
  // Smart pointer:
  T* operator->() const {
    if(!p) return 0;
    return p->data;
  }
  T* current() const {
    if(!p) return 0;
    return p->data;
  }
  // int conversion for conditional test:
  operator int() const { return p ? 1 : 0; }
};
#endif // TSTACK_H_
```

我们可以注意到，这个类已被修改，能支持所有权处理，因为该类可以识别出确切的类型（或至少是基类型，它在运作中使用了虚析构函数）。如同tstash的情形一样，缺省方式是包容器销毁它的对象，但我们可以通过修改析构函数的参数或者通过用 Owns()对成员函数进行读写以改变这种缺省方式。

循环子是非常简单的小规模指示器。当创建一个 tstackIterator时，它从链表的头开始，在链表中只能向前推进。假若打算重新从头部启动，可以创建一个新的循环子；假若要记住链表中的某位置，可以从指向该位置的已生成的循环子处创建一个新的循环子（使用拷贝构造函数）。

为了对循环子所指的对象调用函数，我们可以使用灵巧指针（循环子的通常方法）或使用被称为current()的函数，该函数看上去和灵巧指针相同，因为它返回一个指向当前对象的指针，但它们是不同的，因为灵巧指针执行逆向引用的外部层次（见第 11章）。最后，operator int()指出，是否我们已处在表的尾部和是否允许在条件语句中使用该循环子。

完整的实现包含在这个头文件中，所以这里没有单独的 CPP文件。下面是循环子检测和练习的小例子：

```
//: TSTKTST.CPP -- Use template list & iterator
#include "..\14\tstack.h"
#include "..\11\strings.h"
#include <fstream.h>
#include "..\allege.h"

main() {
  ifstream file("tstktst.cpp");
  allegefile(file);
  const bufsize = 100;
  char buf[bufsize];
  tstack<String> textlines;
  // Read file and store lines in the list:
  while(file.getline(buf,bufsize))
    textlines.push(String::make(buf));
  int i = 0;
  // Use iterator to print lines from the list:
  tstackIterator<String> it(textlines);
  tstackIterator<String>* it2 = 0;
  while(it) {
    cout << *it.current() << endl;
    it++;
    if(++i == 10) // Remember 10th line
      it2 = new tstackIterator<String>(it);
  }
  cout << *(it2->current()) << endl;
  delete it2;
}
```

tstack被实例化为存放String对象并且填充了来自某文件的一些行。然后循环子被创建，用

来在被链接的表中移动。第十行用拷贝构造函数由第一个循环子产生第二循环子，以后，这一行被打印，动态创建的循环子被销毁。这里，动态对象的创建被用于控制对象的生命周期。

这和先前的stack类测试例子十分相似，但是现在所包含的对象能随 tstack的销毁而适当地被销毁。

15.5 字符串和整型

为了在本章的剩余部分进一步改进这些例子，有必要引入功能强大的字符串类，它与整数对象一起来保证初始化。

15.5.1 栈上的字符串

这里是一个更加完全的字符串类，在这本书之前该类已被使用。另外，它使用了模板，添加了一个特殊的特性：对SString实例化时，我们能够决定它存在于堆上还是栈上。

```
//: SSTRING.H -- Stack-based string
#ifndef SSTRING_H_
#define SSTRING_H_
#include <string.h>
#include <iostream.h>

template<int bsz = 0>
class SString {
  char buf[bsz + 1];
  char* s;
public:
  SString(const char* S = "") : s(buf) {
    if(!bsz) { // Make on heap
      s = new char[strlen(S) + 1];
      strcpy(s, S);
    } else { // Make on stack
      buf[bsz] = 0; // Ensure 0 termination
      strncpy(s, S, bsz);
    }
  }
  SString(const SString& rv) : s(buf) {
    if(!bsz) { // Make on heap
      s = new char[strlen(rv.s) + 1];
    strcpy(s, rv.s);
  } else { // Make on stack
    buf[bsz] = 0;
    strncpy(s, rv.s, bsz);
  }
}
SString& operator=(const SString& rv) {
  // Check for self-assignment:
  if(&rv == this) return *this;
```

```
    if(!bsz) { // Manage heap:
      delete s;
      s = new char[strlen(rv.s) + 1];
    }
    // Constructor guarantees length < bsz:
    strcpy(s, rv.s);
    return *this;
  }
  ~SString() {
    if(!bsz) delete []s;
  }
  int operator==(const SString& rv) const {
    return !stricmp(s, rv.s);
  }
  int operator!=(const SString& rv) const {
    return stricmp(s, rv.s);
  }
  int operator>(const SString& rv) const {
    return stricmp(s, rv.s) > 0;
  }
  int operator<(const SString& rv) const {
    return stricmp(s, rv.s) < 0;
  }
  char* str() const { return s; }
  friend ostream&
    operator<<(ostream& os,
               const SString<bsz>& S) {
      return os << S.s;
    }
  };

  typedef SString<> Hstring; // Heap string
  #endif // SSTRING_H_
```

通过使用typedef Hstring，我们可以得到一个普通的基于堆的字符串（使用 typedef而不是使用继承是因为继承需要重新创建构造函数和重载符 =）。但是，假若关心的是生成和销毁许多字符串时的效率，我们可以冒险设定所涉及问题的解的字符最大可能长度。如给出了模板的长度参数，它可以自动地在栈上而不是在堆上创建对象，这意味着每个对象的 new和delete的开销将被忽略。我们能发现运算符=也被提高了运行速度。

字符串的比较运算符使用了称之为 stricmp()的函数，虽然它不是标准 C函数，但却能为大多数编译器的库所认可。它执行字符串比较时会忽略字母大小写。

15.5.2 整型

类integer的构造函数会赋零值，它包含一个自动将类型转换成 int型的运算符，所以可以容

易地提取数据：

```
//: INTEGER.H -- Int wrapped in a class
#ifndef INTEGER_H_
#define INTEGER_H_
#include <iostream.h>

class integer {
  int i;
public:
  // Guaranteed zeroing:
  integer(int ii = 0) : i(ii) {}
  operator int() const { return i; }
  const integer& operator++() {
    i++;
    return *this;
  }
  const integer operator++(int) {
    integer returnval(i);
    i++;
    return returnval;
  }
  integer& operator+=(const integer& x) {
    i += x.i;
    return *this;
  }
  friend ostream&
  operator<<(ostream& os, const integer& x) {
      return os << x.i;
  }
};
#endif // INTEGER_H_
```

虽然这个类相当小（它仅仅满足本章的需要），但我们可以方便地遵循第 11章中的例子而添加很多我们所需要的运算。

15.6 向量

虽然tstash的表现有一点和向量类似，但是创建一个和向量一样的类是很方便的，也就是，它的仅有的行为是索引。（因为它仅有的接口是operator[].）

15.6.1 “无穷”向量

下面的向量类仅仅拥有指针。它从不需要调整大小：我们可以简单地对任何位置进行索引，这些位置可魔术般地变化，而无需提前通知向量类。 operator[]能返回一个指针的引用，所以可以出现在=的左边（它可以是一个左值，也可以是一个右值）。向量类仅仅与指针打交道，所以它工作的对象没有类型限制，对于类型的行为没有事先假定的必要。

下面是它的头文件：

```
//: VECTOR.H -- "Infinite" vector
#ifndef VECTOR_H_
#define VECTOR_H_
#include <stdlib.h>
#include "..\allege.h"

template<class T>
class vector {
  T** pos;
  int pos_sz;
  T** neg;
  int neg_sz;
  int owns;
  enum {
    chunk = 20, // Min allocation increase
    esz = sizeof(T*), // Element size
  };
  void expand(T**& array,int& size,int index);
public:
  vector(int Owns = 1);
  ~vector();
  T*& operator[](int index);
  int Owns() const { return owns; }
  void Owns(int newOwns) { owns = newOwns; }
};
template<class T>
vector<T>::vector(int Owns)
  : pos(0), pos_sz(0),
    neg(0), neg_sz(0),
    owns(Owns) {}

template<class T>
vector<T>::~vector() {
  if(owns)
    for(int i = 0; i < pos_sz; i++)
      delete pos[i];
  free(pos);
  if(owns)
    for(int j = 0; j < neg_sz; j++)
      delete neg[j];
  free(neg);
}

template<class T>
T*& vector<T>::operator[](int index) {
```

```
    if(index < 0) {
      index *= -1;
      if(index >= neg_sz)
        expand(neg, neg_sz, index);
      return neg[index];
    }
    else { // Index >= 0
      if(index >= pos_sz)
        expand(pos, pos_sz, index);
      return pos[index];
    }
  }

  template<class T> void
  vector<T>::expand(T**& array, int& size,
                    int index) {
    const newsize = index + chunk;
    const increment = newsize - size;
    void* v = realloc(array, newsize * esz);
    allegemem(v);
    array = (T**)v;
    memset(&array[size], 0, increment * esz);
    size = index + chunk;
  }
  #endif // VECTOR_H_
```

为了易于实现，向量被分成正和负两个部分，我们可以出于某些原因改变下面的实现使相邻的存储空间得以利用。

当增加存储单元时，将使用私有函数expand()。expand()采用的参数是引用T**&而不是一个指针。这是因为在realloc()之后，它必须改变外部参数以指向一个新的物理地址。另外，还需要参数size和index，以便于对新的单元赋零。（这一点是重要的，因为如果向量拥有对象，析构函数会对所有的指针调用delete。）size以int&进行传递，因为它也必须改变以反映新的存储长度。

在operator[]中，不管是向量内的正或负的部分，若索取一个在当前使用的存储空间之外的存储位置，那么更多的存储空间会被分配。这比内置的数组更加方便，并且创建也无需为存储长度的大小而担心。

```
  //: VECTOR.CPP -- Test "infinite" vector
  #include <fstream.h>
  #include "..\allege.h"
  #include "..\14\vector.h"
  #include "..\14\sstring.h"
  typedef SString<40> String;

  main() {
  ifstream source("vector.cpp");
  allegefile(source);
  const bsz = 255;
```

```
  char buf[bsz];
  vector<String> words;
  int i = 0;
  while(source.getline(buf, bsz)) {
    char* s = strtok(buf, " \t");
    while(s) {
      words[i++] = new String(s);
      s = strtok(0, " \t");
    }
    words[i++] = new String("\n");
  }
  for(int j = 0; words[j]; j++)
    cout << *words[j] << ' ';
}
```

本程序使用了标准C函数strtok()，它取字符缓冲区的起始地址（第一个参数）和寻找定界符（第二个参数）。它用零来代换定界符并返回以标志为起始的地址。假若在随后的时间以第一参数为零的形式来调用它，它将从剩余的字符串中继续抽取直至最后。在上面的例子中，是以空格和制表符为定界符来抽取字词的。每个字词被返回进 String中，而每个指针被存放进words向量中，它最终能以拆分成字词的方式而包含整个文件。

15.6.2 集合

一个集合的约束条件是它的元素不重复。我们可以向一个集合添加元素，也可以测试一下某个元素是否是集合中的成员。下面的集合类使用了一个包含其元素的向量：

```
//: SET.H -- Each entry in a set is unique
#ifndef SET_H_
#define SET_H_
#include "..\14\vector.h"
#include <assert.h>

template<class Type>
class set {
  vector<Type> elem;
  int max;
  int lastindex; // Efficiency tool
  int within(const Type& e) {
    // Requires Type::operator== :
    for(lastindex = 0; lastindex < max;
        lastindex++)
      if(elem[lastindex]->operator==(e))
        return lastindex;
    return -1;
  }
  // Prevent assignment & copy-construction:
```

```
  void operator=(set&);
  set(set&);
public:
  set() : max(0), lastindex(0) {}
  void add(const Type&);
  int contains(const Type&);
  // Where is it in the set?:
  int index(const Type& e);
  Type& operator[] (int index) {
    // No check for shipping application:
    assert(index >= 0 && index < max);
    return *elem[index];
  }
  int length() const { return max; }
};

template<class Type> void
set<Type>::add(const Type& e) {
  if(!contains(e)) {
    elem[max] = new Type(e);//Copy-constructor
    max++;
  }
}

template<class Type> int
set<Type>::contains(const Type& e) {
  return within(e) != -1;
}

template<class Type> int
set<Type>::index(const Type& e) {
  // Prevent a new search if possible:
  if(elem[lastindex]->operator!=(e)) {
    int ind = within(e);
    assert(ind != -1); // Must know it's inside
  }
  return lastindex;
}
#endif // SET_H_
```

add()在检测确定一个新元素不在集合后，将其追加入集合中。 contains()告诉我们对象是否已存在于集合之中， index()告诉我们对象在集合之中的位置。可以使用 operator[]来检索它。length()告诉我们集合中有多少个元素。

上面的例子中始终记着数组元素的数量。作为提高的手段，最后被检索的索引元素会被保存下来，所以index()直接跟随在contains()之后将不会有两次对集合的遍历运算。内联的私有函数within()使实施更为容易。

下面的验证例子使用了集合类而生成一个字词索引，它由存于某文件中的一批字词组成。

```
//: SETTEST.CPP -- Test the "set" class
// Creates a concordance of text words
#include <fstream.h>
#include "..\14\set.h"
#include "..\14\sstring.h"
#include "..\allege.h"
const char* delimiters =
  " \t;()\"<>:{}[]+-=&*#.,/\\"
  "0123456789";

typedef SString<40> String;

main(int argc, char* argv[]) {
  allege(argc == 2, "need file argument");
  ifstream in(argv[1]);
  allegefile(in);
  ofstream out("settest.out");
  set<String> concordance;
  const sz = 255;
  char buf[sz];
  while(in.getline(buf, sz)) {
    // Capture individual words:
    char* s = strtok(buf, delimiters);
    while(s) {
      // Contains 1 entry per unique word:
      concordance.add(s); // Auto type conv.
      s = strtok(0, delimiters);
    }
  }
  for(int i = 0; i < concordance.length(); i++) {
    out << concordance[i] << endl;
  }
}
```

这个程序再次使用了strtok()，但这次的定界符则更多，此外，亦删除了尾部字符及其编号。

注意到由于 add()所希望接受的对象是字符串型的，而对 char*型的类型转换是由使用SString<40>构造函数来自动完成的，该构造函数会取得一个 char*型参数。这样会产生一个临时对象，该对象的地址可传递给 add()，假若add()在表中未发现该临时对象，就会复制它将其加入表中。add()的参数传递使用了对象引用而非指针的方式，这是重要的一点，因为在这里指针没有必要置于其中，假若使用new及运算指针将会最终失去指针。

15.6.3 关联数组

一个普通的数组使用一个整型数值作为某类型序列元素的下标。在一般情况下，若想使用

任意类型为数组下标和其他任意类型为元素，这就是模板的一个理想情形。关联数组可以使用任何类型作为元素的下标。

这里展示的关联数组使用了集合类和向量类创建了关于指针的两个数组：一个为输入类型，另一个为输出类型。假若采用了一个以前未遇到的下标，它会创建一个该下标的副本（假定输入类型具有拷贝构造函数和operator=）作为新的输入对象并且生成一个新的使用缺省构造函数的输出对象。operator[]仅仅返回一个引用给输出对象，所以我们可以使用它来完成运算。也可以用一整型参数调用in_value() 和out_value()函数去产生输入和输出的数组的所有元素：

```
//: ASSOC.H -- Associative array
#ifndef ASSOC_H_
#define ASSOC_H_
#include "..\14\set.h"
#include <assert.h>

template<class In, class Out>
class assoc_array {
  set<In> inVal;
  vector<Out> outVal;
  int max;
  // Prevent assignment & copy-construction:
  void operator=(assoc_array&);
  assoc_array(assoc_array&);
public:
  assoc_array() : max(0) {}
  Out& operator[](const In&);
  int length() const { return max; }
  In& in_value(int i) {
    // No check for shipping application:
    assert(i >= 0 && i < max);
    return inVal[i];
  }
  Out& out_value(int i) {
    assert(i >= 0 && i < max);
    return *outVal[i];
  }
};

template<class In, class Out> Out&
assoc_array<In,Out>::operator[](const In& in){
  if(!inVal.contains(in)) {
    inVal.add(in); // Copy-constructor
    outVal[max] = new Out; // Default constr.
    max++;
  }
  int x = inVal.index(in);
  return *outVal[inVal.index(in)];
```

```
}
#endif // ASSOC_H_
```

关联数组的一个经典的程序实例是对一文件进行字数统计，这是一个相当简单但很实用的工具。关联数组的输出值不能是内部数据类型（built-in），这是关联数组的限制之一。因为内部数据类型没有缺省构造函数，在创建时不能初始化。为了在字数统计程序中解决该问题，整数类和SString类一同用于关联数组的检测：

```
//: ASSOC.CPP -- Test of associative array
#include "..\14\assoc.h"
#include "..\14\sstring.h"
#include "..\14\integer.h"
#include "..\allege.h"
#include <fstream.h>
#include <ctype.h>

main() {
  const char* delimiters =
    " \t;()\"<>:{}[]+-=&*#.,/\\";
  assoc_array<SString<80>, integer> strcount;
  ifstream source("assoc.cpp");
  allegefile(source);
  ofstream out("assoc.out");
  allegefile(out);
  const bsz = 255;
  char buf[bsz];
  while(source.getline(buf, bsz)) {
    char* s = strtok(buf, delimiters);
    while(s) {
      strcount[s]++; // Count word
      s = strtok(0, delimiters);
    }
  }
  for(int i = 0; i < strcount.length(); i++) {
    out << strcount.in_value(i) << " : "
        << strcount.out_value(i) << endl;
  }

  // The "shopping list" problem:
  assoc_array<SString<>, integer> shoplist;
  ifstream list("shoplist.txt");
  allegefile(list);
  ofstream olist("shoplist.out");
  allegefile(olist);
  while(list.getline(buf, bsz)) {
    int i = strlen(buf) - 1; // Last char
```

```
    while(isspace(buf[i]))
      i--; // Find nonzero char at end
    while(!isspace(buf[i]))
      i--; // Back up to space or tab
    int count = atoi(&buf[i+1]); // Use value
    while(isspace(buf[i]))
      i--; // Back up to non-whitespace
    buf[i+1] = 0; // Mark end of description
    i = 0;
    while(isspace(buf[i]))
      i++; // Find start of first word
    shoplist[&buf[i]] += count;
  }
  for(int j = 0; j < shoplist.length(); j++) {
    olist << shoplist.in_value(j) << " : "
          << shoplist.out_value(j) << endl;
  }
}
```

该程序中关键的一行是：

```
strcount[s]++;//count word
```

由于s是char*型参数，而operator[]所希望的是SString<80>型，编译器可生成一个临时对象传递给使用构造函数SString<80>(char*)的operator[]。该临时对象被创建于栈上，所以它可以快速生成和销毁。

operator[]返回一个integer&给assoc_array中相应的对象，这样我们可以赋于对象任何行为。这里，简单地通过累加运算来说明发现了另一个单词。主程序的后面部分用于解决传统“货单”问题。货单文件的每一行都包含项目（名称可能用空格分开）和数量。这些项目可能在表中不止一次出现，关联数组可对其进行统计。在这些代码中 C的标准宏isspace()的使用作为对环境空间的额外补偿。下面代码行用于统计的实现：

```
shoplist[&buf[i]] +=count;
```

和以前一样， char*的内部检索会产生一个SString对象，因为这是索引运算符预期搜寻的对象，类型间的转化由构造函数完成。如果我们跟踪该程序就会发现由这个临时对象引起的构造函数和析构函数的调用。

15.7 模板和继承

没有东西能妨碍我们采用在普通类中的方法来使用类模板。例如我们能很容易地从一个模板中获得继承，可以从已存在的模板中继承和加以实例化从而创建一个新的模板。尽管 tstash类在前述中已经可满足我们的要求，现在当我们希望它追加自排序功能时，可以方便地对其进行代码重用和数值添加：

```
//: SORTED.H -- Template inheritance
#ifndef SORTED_H_
#define SORTED_H_
#include <stdlib.h>
```

```
#include <string.h>
#include <time.h>
#include "..\14\tstash.h"
#include "..\14\set.h"
template<class T>
class sorted :  public tstash<T> {
  void bubblesort();
public:
  int add(T* element) {
    tstash<T>::add(element);
    bubblesort();
    return 0; // Sort moves the element
  }
};
template<class T>
void sorted<T>::bubblesort() {
  for(int i = count(); i > 0; i--)
    for(int j = 1; j < i; j++)
      if(*storage[j-1] > *storage[j]) {
        // Swap the two elements:
        T* t = storage[j-1];
        storage[j-1] = storage[j];
        storage[j] = t;
      }
}

// Quick & dirty sorted set:
template<class T>
class sortedSet : public set<T> {
  sorted<T> Sorted;
public:
  void add(T& e) {
    if(contains(e)) return;
    set<T>::add(e);
    Sorted.add(new T(e));
  }
  T& operator[] (int index) {
    assert(index >= 0 && index < length());
    assert(Sorted[index]);
    return *Sorted[index];
  }
  int length() {
    return Sorted.count();
  }
};
```

```
// Unique random number generator:
template<int upper_bound>
class urand {
  int map[upper_bound];
  int recycle;
public:
  urand(int Recycle = 0);
  int operator()();
};

template<int upper_bound>
urand<upper_bound>::urand(int Recycle = 0)
  : recycle(Recycle) {
  memset(map, 0, upper_bound * sizeof(int));
  // Seed the random number generator:
  time_t t;
  srand((unsigned)time(&t));
}

template<int upper_bound>
int urand<upper_bound>::operator()() {
  if(!memchr(map, 0, upper_bound)) {
    if(recycle)
      memset(map, 0,
             sizeof(map) * sizeof(int));
    else
      return -1; // No more spaces left
  }
  int newval;
  while(map[newval = rand() % upper_bound])
    ; // Until unique value is found
  map[newval]++; // Set flag
  return newval;
}
#endif // SORTED_H_
```

本例子包含了一个随机数发生器类，它可产生唯一数和重载 operator()以便使用一般的函数调用语句。urand的唯一性是由保存随机数空间的所有可能的数的影（map）象而产生的（随机数空间的上界由模板参数设置），并标记每一个已使用的为关闭。构造函数的第二个可选参数的作用是允许我们在界内随机数用完的情况下可以重用这些数。注意，为了优化运行速度我们将影象定义成固定完整数组，而不论我们需要多少数。假若我们打算优化数组长度，可以这样改动下面的实现：把map安排成动态申请存储方式；把随机数本身送入 map而不是置标志，这样的改变不会影响任何客户代码。

模板sorted为所有由它实例化而产生的类施加一个约束：它们必须包含一个 >运算符。在SString中，这种施加是明显的，但是在 integer中，自动类型转换运算符 int()提供了一个内置>

运算符的途径。当模板提供更多的功能时，通常要对类赋予更多的需求，。有时不得不继承被包含的类以增加必要的功能。注意使用重载运算符的价值： integer类所提供的功能依赖于它的底层实现。

在例中，可以看到在tstash中的storage以保护方式而非私有方式定义的好处。这对于让类sorted知道很重要，这是真正的依赖性。假若改变 tstash的下层实现的一些东西而非一个数组，如链表，冒泡排序中的元素交换就会和原来完全不同，于是从属类 sorted也需要改变。然而，一个更好的选择是（假若有可能的话），对让tstash实现采用保护方法的一种更好的替代是提供足够的保护接口函数，这样一来，访问和交换可在派生类中完成而无需涉及底层实现。这种方法仍然可以改变下层实现，但不会传播这些修改。

注意，附加类sortedSet说明怎样可以快速地由已存在的类中粘贴所需的功能。 sortedSet可从set类中获取接口，而且当添加一个新元素到set类时，也同时为其添加这个元素到sorted对象，所返回的值全是排序过的。我们可能会考虑设计一个更为简化、有效的类版本，在这里就不再详述了（标准C++模板库包含一个可自排序的集合类）。下面是对SORTED.H的测试：

```
//: SORTED.CPP -- Testing template inheritance
#include "..\14\sorted.h"
#include "..\14\sstring.h"
#include "..\14\integer.h"
typedef SString<40> String;

char* words[] = {
  "is", "running", "big", "dog", "a",
};
const wordsz = sizeof words / sizeof *words;

main() {
  sorted<String> ss;
  for(int i = 0; i < wordsz; i++)
    ss.add(new String(words[i]));
  for(int j = 0; j < ss.count(); j++)
    cout << ss[j]->str() << endl;
  sorted<integer> is;
  urand<47> rand1;
  for(int k = 0; k < 15; k++)
    is.add(new integer(rand1()));
  for(int l = 0; l < is.count(); l++)
    cout << *is[l] << endl;
}
```

该例通过创建一个排序数组来验证SString 和interger类。

15.7.1 设计和效率

在sorted中，每次调用add()时，新元素都将被插入，数组也重新排序。这里使用的冒泡排序方法效率低下，不提倡使用（但它易于理解和编码）。由于冒泡排序方法是私有实现的一部分，在这里是相当合适的。在我们开发程序时，一般的步骤是：

1) 使类接口正确。
2) 尽量迅速、准确地实现原型。
3) 验证我们的设计。

常常，仅当我们集成工作系统的初期“草稿”时才会发现类接口问题，这种情况非常普遍。在系统集成和初次实现期间，我们还可能发现需要“帮助者”类，如同对包容器和循环子的需要一样。有时在系统分析期间很难发现上述问题（在分析中我们的目标是得到能快速实现和检测的全貌设计）。只有在设计被验证后，我们才有必要花时间对其进行完全刷新和考虑性能要求。假如设计失败或者性能要求不需考虑，则冒泡排序方法就不错了，没有必要进一步浪费时间。（当然，一个理想的解决方案是利用别人的已被证实的排序包容器；首先应留意标准C++的模板库）。

15.7.2 防止模板膨胀

每次模板实例化，其中的代码都会重新生成（其中的内联函数除外）。假若模板内的一些功能并不依赖定义类型，我们可以把它们放入一个通用基类以避免不必要的代码重新生成。例如在第13章的INHSTAK.CPP(见13.5.2)中的继承被用于定义stack所能接受和产生的类型。下面是一个模板化的版本代码：

```
//: NOBLOAT.H -- Templatized INHSTAK.CPP
#ifndef NOBLOAT_H_
#define NOBLOAT_H_
#include "..\11\stack11.h"

template<class T>
class nbstack : public stack {
public:
  void push(T* str) {
    stack::push(str);
  }
  T* peek() const {
    return (T*)stack::peek();
  }
  T* pop() {
    return (T*)stack::pop();
  }
  ~nbstack();
};

// Defaults to heap objects & ownership:
template<class T>
nbstack<T>::~nbstack() {
  T* top = pop();
  while(top) {
    delete top;
    top = pop();
  }
}
#endif // NOBLOAT_H_
```

在以前，内联函数不产生代码，而是通过仅一次性地创建一个基类代码提供其功能。但是所有权问题则可以通过加入一个析构函数来解决 (它依赖类型，必须由模板来构造)，在这里的所有权是缺省的。注意，当基类析构函数被调用时，栈将被清空，所以不会出现重复释放问题。

15.8 多态性和包容器

多态性、动态对象生成和包容器在一个真正的面向对象程序中和谐地被利用，这是很普遍的。动态对象生成和包容器所解决的问题在于设计初期我们可能不知道需要多少对象，需要什么类型的对象，这是因为包容器可持有指向基类对象的指针，每逢我们把派生类指针放入包容器，会发生向上映射（具有相应的代码组织和可扩展性的好处）。下面的例子有点像垃圾回收的工作过程，首先所有的垃圾被放入一个垃圾箱中，然后分类放入不同的箱中，它有一个函数用于遍历垃圾箱并估算出其中什么有价值。这里的垃圾回收模拟实现并不完美，在第 18章说明“运行时类型识别(RTTI) ”时，再对该例进一步介绍。

```
//: RECYCLE.CPP -- Containers & polymorphism
#include <fstream.h>
#include <stdlib.h>
#include <time.h>
#include "..\14\tstack.h"
ofstream out("recycle.out");

enum type { Aluminum, Paper, Glass };

class trash {
  float Weight;
public:
  trash(float Wt) : Weight(Wt) {}
  virtual type trashType() const = 0;
  virtual const char* name() const = 0;
  virtual float value() const = 0;
  float weight() const { return Weight; }
  virtual ~trash() {}
};

class aluminum : public trash {
  static float val;
public:
  aluminum(float Wt) : trash(Wt) {}
  type trashType() const { return Aluminum; }
  virtual const char* name() const {
    return "aluminum";
  }
  float value() const { return val; }
  static void value(int newval) {
    val = newval;
```

```
  }
};

float aluminum::val = 1.67;

class paper : public trash {
  static float val;
public:
  paper(float Wt) : trash(Wt) {}
  type trashType() const { return Paper; }
  virtual const char* name() const {
    return "paper";
  }
  float value() const { return val; }
  static void value(int newval) {
    val = newval;
  }
};

float paper::val = 0.10;

class glass : public trash {
  static float val;
public:
  glass(float Wt) : trash(Wt) {}
  type trashType() const { return Glass; }
  virtual const char* name() const {
    return "glass";
  }
  float value() const { return val; }
  static void value(int newval) {
    val = newval;
  }
};

float glass::val = 0.23;

// Sums up the value of the trash in a bin:
void SumValue(const tstack<trash>& bin,ostream& os){
  tstackIterator<trash> tally(bin);
  float val = 0;
  while(tally) {
    val += tally->weight() * tally->value();
    os << "weight of " << tally->name()
       << " = " << tally->weight() << endl;
```

```
    tally++;
  }
  os << "Total value = " << val << endl;
}

main() {
  // Seed the random number generator
  time_t t;
srand((unsigned)time(&t));

tstack<trash> bin; // Default to ownership
// Fill up the trash bin:
for(int i = 0; i < 30; i++)
  switch(rand() % 3) {
    case 0 :
      bin.push(new aluminum(rand() % 100));
      break;
    case 1 :
      bin.push(new paper(rand() % 100));
      break;
    case 2 :
      bin.push(new glass(rand() % 100));
      break;
  }
// Bins to sort into:
tstack<trash> glassbin(0); // No ownership
tstack<trash> paperbin(0);
tstack<trash> ALbin(0);
tstackIterator<trash> sorter(bin);
// Sort the trash:
// (RTTI offers a nicer solution)
while(sorter) {
  // Smart pointer call:
  switch(sorter->trashType()) {
    case Aluminum:
      ALbin.push(sorter.current());
      break;
    case Paper:
      paperbin.push(sorter.current());
      break;
    case Glass:
      glassbin.push(sorter.current());
      break;
  }
  sorter++;
```

```
    }
    SumValue(ALbin, out);
    SumValue(paperbin, out);
    SumValue(glassbin, out);
    SumValue(bin, out);
}
```

这里使用了基类中的虚函数结构，这些函数将在派生类中得到再定义。由于包容器 tstack是trash的实例化，所以它含有trash指针，这些指针是指向基类的。然而， tstack也会含有指向trash的派生类对象的指针，正如调用push()所看到的那样。随着这些指针的加入，它们会失去本来的特定身份而变成trash的指针。然而由于多态性，当通过tally 和 sorter循环调用虚函数时，与要求相适应的行为仍会发生。（注意，使用循环子的灵巧指针会导致虚函数的调用。）

trash类还包含一个虚析构函数，向任何类中自动增加内容都应当利用虚函数。当bin包容器超出了相应范围时，包容器的析构函数会为它所包容的所有对象调用虚析构函数，进行有效的清除。

由于包容器类模板一般很少有所见到的“普通”类的向下继承和向上映射，所以我们几乎不会看到在这些类中存在虚函数，它们的重用是以模板方式而非继承方式。

15.9 包容器类型

这里所采用的一系列包容器类大约和“数据结构”类工具相当（当然包容器由于具备一些相关联的功能因而比数据结构更丰富）。包容器将其功能和数据结构打包集成在一起，它能更为自然地表达“数据结构”的概念。

虽然包容器可能需要特定的行为（如在栈尾压入和抛出元素），但是通过使用较通用的循环子便可获得更大的自由度。下列的大多数类型都很容易支持关联循环子。

所有的包容器都允许放入和取出一些东西。它们的不同在于其功能用途，而有一些包容器的相异之处则仅仅在于存放内容的类型不同。它们的不同在于访问速度：一些易于线性访问，但在序列中间插入元素时，时间开销却较高。其他的一些在中间插入元素时时间开销较低，但线性访问的开销却较高。如果要在这两种情形下做出取舍，应首先着眼于最可通融的方法。如果在程序运行后，发现速度的通融性较差则需要进一步优化，优化成更加高效的方法。

下面是存在于C++标准模板库STL（standard template library）中的包容器子集， STL将在附录A中讨论：

袋子：项目的（可能有重复）集合。它的元素没有特定的次序要求， STL不包含它，这是因为其功能可由表和向量来实现。

集合：不允许有重复元素的袋子就是集合。在 STL中，集合是以一种联合包容器(associative container)来描述的，联合包容器可以根据关键字向包容器提供和从包容器中取回数据元素。在集合中数据元素的存取检索关键字就是元素本身。一个 STL的多重集合允许将同一关键值的许多副本放入不同的对象中。

向量：可索引的项目序列。由于它有一致的访问时间，所以可作为缺省选择。

队列：从尾部追加元素，从头部删除元素的项目序列。它是双端队列的子集，有时不实现它们，但我们可以用双端队列来代替。

双端队列：具有两个端部的队列。序列中项目元素的追加和删除可从任意一端进行。在大部分的插入和删除运算发生在序列头部或尾部时，可用其代替列表以提高效率。在头部或尾部做插入和删除运算时，双端队列的时间复杂度为常量级，但在序列中部实施运算时呈线性时间

复杂度。

栈：在相同一端实施追加和删除的项目序列。尽管在本书中被用作一个例子，其实它是双端队列功能的一个子集。

环形队列：环形结构的项目序列，元素的追加和删除位于环形结构的顶端，它是头部和尾部关联的队列。通常让其支持系统底层活动，非常有效而且其时间复杂度为常量级。例如在一个通讯中断服务例程中，插入一些字符而随后将其删除，将不用担心存储的耗尽及必须的时间分配。但是，如不加以细致地编程，环形队列会超出正常的控制范围。 STL不包含环形队列。

列表：允许以相等的时间在任意点实施插入和删除的有根的项目序列。它不提供快速随机访问，但能在序列中间进行快速插入和删除运算。列表一般以单链表和双链表的形式来实现。单链表的遍历是单方向的，而双链表可在任意节点上向前向后移动。由于单链表仅包含一个指向下一个节点的指针而双链表则包含前趋和后继两个指针，所以单链表较双链表的存储开销低。但是在插入和删除工作效能上，双链表则优于单链表。

字典：关键字和相应值的映射，在STL中被称为“ 映像 ”（它是联合包容器的另一种形式）。关键字和相应值的映射对有时被称为“ 联系 ”。字典值的存取和关键字是相关联的。 STL也提供多重映射，允许多重映射相同关键字到相应值的多个副本上，这和哈希表相当。

树：存在一个根节点的节点和弧（节点间的连接）的集合。树不包含环（没有封闭的路径）和交叉路径。STL中不提供树，树具有的特性功能由STL的其他类型来提供。

二叉树：一个普通树从每个节点上射出的弧的数目是没有限制的，而二叉树从每个节点上最多只射出两条弧，即“ 左 ”“ 右 ”两条弧。由于节点的插入位置随其值而定，当搜寻所期望值时则不必浏览许多节点（而线性表则不同），所以二叉树是最快的信息检索方法之一。平衡二叉树在每次插入节点时，它重新组合以保证树每一部分的深度都不超过基准值。然而每次插入时的平衡处理的开销则较高。

图：无根节点的节点和弧的集合。图可以包含环和交叉路径。它通常更具有理论上的意义并不常用于实际实施。它不是STL的一部分。

• 不要做重复工作

在工作中我们首先应在编译器或其他方便之处寻求 STL中的公共组件，或从第三方供应商处购得，以减少重复工作量。从头创建这些组件目的仅仅是作为练习或没有办法的办法。

15.10 函数模板

类模板描述了类的无限集合，出现模板的大部分地方都是出现类的地方。 C++同样可以支持函数的无限集合的概念，有时它非常有用，其语法部分除用函数来替代类外和类模板没有什么两样。

假若打算创建一些函数，这些函数除了处理各自的不同类型外，函数体看上去都相同，这就有必要创建一个函数模板来描述这些函数。函数模板的典型例子是一个排序函数 [1]，然而它可适用于各种场合，下面的第一个例子作为示范。第二个例子则揭示函数模板连同循环子和包容器中的使用。

15.10.1 存储分配系统

例程malloc()、calloc()和 realloc()都可以较安全地对未开辟的存储空间进行分配。下面的

[1] 参看作者的C++Inside & out(Osborne/McGraw-Hill, 1993)。

函数模板可产生既能分配一部分新的存储空间又能为已开辟的区域重设大小（如同 realloc())的函数getmem() 。另外，它仅对新的存储进行清零处理，并且检查被其分配的存储空间。而且，提交给getmem()的参数仅是所期望的某类型元素的数目而非字节数目，所以可以降低程序出错的概率。这是它的头文件：

```
//: GETMEM.H -- Function template for memory
#ifndef GETMEM_H_
#define GETMEM_H_
#include <stdlib.h>
#include <string.h>
#include "..\allege.h"

template<class T>
void getmem(T*& oldmem, int elems) {
  typedef int cntr; // Type of element counter
  const int csz = sizeof(cntr); // And size
  const int Tsz = sizeof(T);
  if(elems == 0) {
    free(&(((cntr*)oldmem)[-1]));
    return;
  }
  T* p = oldmem;
  cntr oldcount = 0;
  if(p) { // Previously allocated memory
    ((cntr*)p)--; // Back up by one cntr
    oldcount = *(cntr*)p; // Previous # elems
  }
  T* m = (T*)realloc(p, elems * Tsz + csz);
  allegemem(m);
  *((cntr*)m) = elems; // Keep track of count
  const cntr increment = elems - oldcount;
  if(increment > 0) {
    // Starting address of data:
    long startadr = (long)&(m[oldcount]);
    startadr += csz;
    // Zero the additional new memory:
    memset((void*)startadr, 0, increment * Tsz);
  }
  // Return the address beyond the count:
  oldmem = (T*)&(((cntr*)m)[1]);
}

template<class T>
inline void freemem(T * m) { getmem(m, 0); }

#endif // GETMEM_H_
```

为了能够仅在新的存储区进行清零处理，将有一个用来指示所分配元素数目的计数器被置于每一存储区的首部。计数器的类型为 typedef cntr ，当处理较大的存储区时可将其由整型改为长整型。（当使用长整型时其他的一些问题会出现，然而不管怎样，编译器都会在警示中提示这些问题。）

之所以使用指针引用 oldmem作为参数是由于外部变量（一个指针）必须改为指向新存储区，oldmem必须指向零（以分配新存储区）或指向由 getmem()创建的存储区。该函数设想我们能正确地使用它，但如果我们打算对其调试，可在计数器附近加一个辅助标识，通过检查该标识来帮助发现getmem()中的错误调用。

如果所要求的元素数为零，则该存储被释放。有另外一个函数模板 freemem（），是这个行为的别名。

我们将注意到， getmem()的处理层次很低，存在许多底层调用和字节处理。例如，oldmem指针并不指向存储区的真实的起始位置，而恰好在起始位置计数器之后。所以在用free()释放存储区时，getmem()必须将指针向后退由cntr占用的存储空间的数目。由于oldmem的类型是T*，必须首先将其映射为cntr*，然后它被向后索引一个位置。最后，为 free()产生指定位置地址的语句表达为：

```
free(&(((cntr*) oldmem )[-1]));
```

同样地，假若这是一个已经分配的存储空间， getmem()必须后退一个cntr长度，以获取真实的存储空间的起始地址并取回先前的元素数目。在 realloc()内部需要真实的起始地址。若要开辟的存储空间是向上增加的，在 memset()中的起始地址和需要清零的元素数目可由新的元素数目减去旧的元素数目而求得。最后，产生计数器后面的地址，并将其赋给 oldmem，赋值语句为：

```
oldmem=(T*)&(((cntr*)m)[1]);
```

再者，由于oldmem是对一个指针的引用，这导致传给getmem()的外部参数的变化。

下面的程序用于测试getmem()。它分配和填入值，然后再进一步开辟更多的存储空间：

```
//: GETMEM.CPP -- Test memory function template
#include "..\14\getmem.h"
#include <iostream.h>

main() {
  int* p = 0;
  getmem(p, 10);
  for(int i = 0; i < 10; i++) {
    cout << p[i] << ' ';
    p[i] = i;
  }
  cout << '\n';
  getmem(p, 20);
  for(int j = 0; j < 20; j++) {
    cout << p[j] << ' ';
    p[j] = j;
  }
  cout << '\n';
  getmem(p, 25);
```

```
  for(int k = 0; k < 25; k++)
    cout << p[k] << ' ';
  freemem(p);
  cout << '\n';

  float* f = 0;
  getmem(f, 3);
  for(int u = 0; u < 3; u++) {
      cout << f[u] << ' ';
      f[u] = u + 3.14159;
    }
    cout << '\n';
    getmem(f, 6);
    for(int v = 0; v < 6; v++)
      cout << f[v] << ' ';
    freemem(f);
  }
```

在每次调用getmem()后，存储区中的值都被打印出来，可以看到新的存储区都被清零了。

注意由整型指针和浮点型指针而实例化getmem()的不同版本。由于上述功能涉及到非常底层的处理，我们可能认为应当使用一个非模板函数并传递 void* &作为oldmem的方式来实现。这种想法是不能实现的，因为编译器必须将我们的类型转化成 void*。为了获取引用，编译器会安排一个临时域，由于修改的是临时指针而非我们真正想要修正的指针，所以会出现错误。故使用函数模板为参数产生相应的确切类型是必需的。

15.10.2 为tstack提供函数

假设我们打算拥有一个 tstack并使一函数适用它所包含的所有对象。由于 tstack可以包含任意类型的对象，所以该函数应该可以在 tstack的任意类型和其包含的任意类型对象下工作：

```
//: APPLIST.CPP -- Apply a function to a tstack
#include "..\14\tstack.h"
#include <iostream.h>

// 0 arguments, any type of return value:
template<class T, class R>
void applist(tstack<T>& tl, R(T::*f)()) {
  tstackIterator<T> it(tl);
  while(it) {
    (it.current()->*f)();
    it++;
  }
}

// 1 argument, any type of return value:
template<class T, class R, class A>
void applist(tstack<T>& tl, R(T::*f)(A), A a) {
```

```
  tstackIterator<T> it(tl);
  while(it) {
    (it.current()->*f)(a);
    it++;
  }
}

// 2 arguments, any type of return value:
template<class T, class R, class A1, class A2>
void applist(tstack<T>& tl, R(T::*f)(A1, A2),
    A1 a1, A2 a2) {
  tstackIterator<T> it(tl);
  while(it) {
    (it.current()->*f)(a1, a2);
    it++;
  }
}

// Etc., to handle maximum probable arguments

class gromit { // The techno-dog
  int arf;
public:
  gromit(int Arf = 1) : arf(Arf + 1) {}
  void speak(int) {
    for(int i = 0; i < arf; i++)
      cout << "arf! ";
    cout << endl;
  }
  char eat(float) {
    cout << "chomp!" << endl;
    return 'z';
  }
  int sleep(char, double) {
    cout << "zzz..." << endl;
    return 0;
  }
  void sit(void) {}
};

main() {
  tstack<gromit> dogs;
  for(int i = 0; i < 5; i++)
    dogs.push(new gromit(i));
  applist(dogs, &gromit::speak, 1);
  applist(dogs, &gromit::eat, 2.0f);
```

```
  applist(dogs, &gromit::sleep, 'z', 3.0);
  applist(dogs, &gromit::sit);
}
```

applist()函数模板可获取包容器类的引用及类中成员函数的指针。为了在栈中移动applist(),这里使用了一个循环子并且把函数应用于每一个对象。假若我们已经忘了成员指针的语法，可复习第10章的后面部分。

我们可以看到有不只一个applist()版本，所以重载函数模板是可行的。虽然它们都可接受任意类型的返回值（这被忽略了，但对于匹配成员指针来说，类型信息则是所要求的），applist()的每一个版本都有一些不同的参数，由于它是一个模板，所以这些参数的类型是任意的。（在类gromit中，可以看到一批不同的函数[1]）。由于不存在“超模板”为我们生成模板，所以我们必须决定究竟需要多少参数。

虽然applist()的定义相当复杂，其中的一部分不能指望一个初学者去理解它，但是它的使用则非常清晰而简单，初学者仅仅需要知晓完成什么而非怎样完成，所以初学者可以容易地使用它。我们应尽量把程序组件分成不同的类别，仅仅关心所要完成的目标而不需关心底层的实现细节。棘手的细节问题仅仅是设计者的任务。

当然，这些功能类型会牢固地联系着tstack类，所以通常我们应该随tstack一道在头文件中找到这些函数模板，加以分析利用。

15.10.3 成员函数模板

把applist()安排为成员函数模板也是可行的，这是一个和类模板相对独立的模板定义，而且它仍然是类的成员。因此能够使用下面更巧妙的语句：

```
dogs.applist(&gromit::sit);
```

这和在类内部引出普通函数的做法（第2章）相类似[2]。

15.11 控制实例

显式实例化一个模板有时是有用的，这会告诉编译器为模板的特定版本安排代码，即使并不打算生成一个对象。为了实现它，可以重用下面的模板关键字：

```
template class bobbin<thread>;
template void sort<char>(char *[]);
```

这是SORTED.CPP例子的一个版本（见15.7），在使用它之前会显式实例化一个模板：

```
//: GENERATE.CPP -- Explicit instantiation
#include "..\14\sorted.h"
#include "..\14\integer.h"
// Explicit instantiation:
template class sorted<integer>;

main() {
  sorted<integer> is;
  urand<47> rand1;
```

[1] 参见对Nick Park的英国活泼短片《糟糕的裤子》。

[2] 检查我们的编译器版本信息，看它是否支持函数模板。

```
  for(int k = 0; k < 15; k++)
    is.add(new integer(rand1()));
  for(int l = 0; l < is.count(); l++)
    cout << *is[l] << endl;
}
```

在该例子中，显式实例并不真正地完成什么事情，它未参与程序的运作。显式实例仅在外部控制的特定情况下才是必须的。

• 模板特殊化

sorted向量仅工作于用户类型的对象上。例如，它不能对 char*型数组进行排序。为了创建一个特定版本，我们可以自己写一个实例版本，就像编译器已经通过并把我们的类型代入了模板参数一样。但是要把我们自己的代码放入特殊化的函数体中。下面的例子揭示char*型sorted向量：

```
//: SPECIAL.CPP -- Template specialization
// A special sort for char*
#include "..\14\sorted.h"
#include <iostream.h>
class sorted<char> :  public tstash<char> {
  void bubblesort();
public:
  int add(char* element) {
    tstash<char>::add(element);
    bubblesort();
    return 0; // Sort moves the element
  }
};

void sorted<char>::bubblesort() {
  for(int i = count(); i > 0; i--)
    for(int j = 1; j < i; j++)
      if(strcmp(storage[j], storage[j-1]) < 0) {
        // Swap the two elements:
        char* t = storage[j-1];
        storage[j-1] = storage[j];
        storage[j] = t;
      }
}

char* words[] = {
  "is", "running", "big", "dog", "a",
};
const wsz = sizeof words/sizeof *words;

main() {
  sorted<char> sc;
  for(int k = 0; k < wsz; k++)
```

```
    sc.add(words[k]);
  for(int l = 0; l < sc.count(); l++)
    cout << sc[l] << endl;
}
```

在bubblesort()中，我们可以看到使用的是 strcmp()而不是>。

15.12 小结

包容器类是OOP的一个基本部分，它是简化和隐藏实施细节，提高开发效率的另一种方法。另外，它通过对C中的旧式数组和粗糙的数据结构技术的更新替代从而大大地提高了灵活性和安全性。

由于包容器是客户程序员所需要的，所以包容器的实质是便于使用，这样，模板就被引入进来。对源代码进行重用（相反的是继承和组合实施对对象代码的重用）的模板语句可对初学者来说变得十分平常。实际上，使用模板实施代码重用比继承和组合容易得多。

虽然在本书中我们已经学习了包容器和循环子，但在实际中，学习编译器所带的包容器和循环子是更迅速的方法，要不然就从第三方供应商处购买一个库 [1]。标准C++库是很完备的，但在包容器和循环子方面并不充分。

本章简单地提及了包容器类设计方面的内容，我们可以加以总结以有更多的体会。一个复杂的包容器类库可能涉及所有的附加内容，包括持久性（在第 16章中介绍）和垃圾回收（在第12章中介绍），也包含处理所有权问题的附加方法。

15.13 练习

1. 修改第14章练习一的结果，以便使用tstack 和tstackIterator替代 shape指针数组。增加针对类层次的析构函数以便在tstack超出范围时观察shape对象被析构。

2. 修改第14章例子SSHAPE2.CPP以使用tstack替代数组。

3. 修改RECYCLE.CPP以使用tstash替代tstack。

4. 改变SETTEST.CPP以使用 sortedSet替代set.。

5. 为 tstash 类复制APPLIST.CPP的功能。

6. 将TSTACK.H拷贝到新的头文件中，并增加 APPLIST.CPP中的函数模板作为tstack的成员函数模板。本练习要求我们的编译器支持成员函数模板。

7. （高级）修改tstack类以促进增加所有权的区分粒度。为每一个链接增加标志以表明它是否拥有其指向的对象，并在 add()函数和析构函数中支持这一标志信息。增加用于读取和改变每一链接所有权的成员函数，并在新的上下文环境中确定 add()标志的含义。

8. （高级）修改tstack类，使每一个入口包含引用计数信息（不是它们所包容的对象)并且增加用于支持引用计数行为的成员函数。

9. （高级）改变SORTED.CPP中urand的底层实现以提高其空间效率（SORTED.CPP后面段落所描述的) 而非时间效率。

10. （高级）将GETMEM.H中的typedef cntr从整型改成长整型，并且修改代码以消除失去精度的警示信息，这是一个指针算术问题。

11. （高级）设计一个测试程序，用于比较创建在堆上和创建在栈上的 SString的执行速度。

---

[1] C++标准库包含一个非常地道的但并非详尽无遗的包容器和循环子集。

第16章 多重继承

多重继承MI（multiple inheritance)的基本概念听起来非常简单。

可以通过继承多个基类来生成一个新类。这种语法正如料想的那样，只要继承图简单，MI也不复杂。然而MI会引入许多二义性和奇异的情况，它贯穿于本章。现在首先给出关于本主题概述。

16.1 概述

在C++以前，最为成功的面向对象语言是Smalltalk。Smalltalk从开始建立就成为面向对象的语言。就面向对象来说，Smalltalk是“纯种的”,而C++则是“混血的”，这是因为C++是在C语言上建立的。Smalltalk的一个设计原则是：所有的类应从一个单一层次上进行派生，它们根植于一个基类（称之为Object，它是基于对象层次的模式），在Smalltalk中不从一个已生成类中进行继承就不能生成一个新类，所以在创建新类之前必须学习 Smalltalk类库，这就是为什么得花相当长的时间才能成为Smalltalk熟手的原因。所以Smalltalk class 类层次总是一棵简单的单一树。

Smalltalk的类有相当部分是共同的，例如其中Object的行为和特性必然是相同的，所以几乎不会碰上需要继承多个基类的情况。然而只要我们想一想， C++可以创建任意个继承树，因而对C++语言的逻辑完备性来说，它必须支持多个类的结合，由此要求有多重继承性。

然而，这并不是多重继承必须存在的令人信服的理由。过去有，现在仍然有关于 MI是C++精华的否定意见。AT&T 的cfront2.0 版首先把MI加入语言并对语言做了明显修改。从那以后，许多改变我们编程方法的其他特性（特别如模板）都被加入进去并且降低了 MI的重要性。我们可以把MI当作语言的“次要”特性。

一个最为紧迫的问题是如何驾驭涉及 MI的包容器。假若我们打算创建一个使每个用户都容易使用的包容器，在包容器内使用 void*是一个方法，如pstash 和 stack。而Smalltalk的方法是创建一个包含 Objects的包容器（记住Object是整个Smalltalk层次的基类型），由于Smalltalk的所有东西都是由Object派生的，所以任何含有Objects的包容器都可以包容任何东西，该方法是不错的。

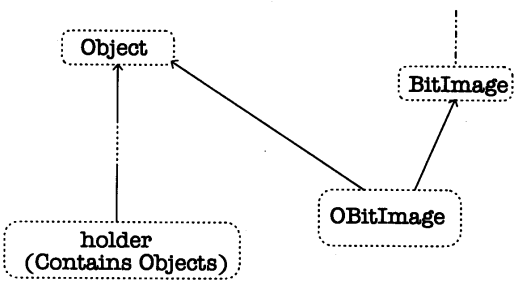


图 16-1

现在考虑C++的情况。假若供应商A创建了一个基于对象的层次结构，该层次包含了一组有用的包容器，其中含有一个打算使用的称之为 holder的包容器。现在我们又发现了供应商 B的类层次，其中有一些重要的其他类，如BitImage类，它可以含有位图。创建一个位图包容器仅有的方法是对BitImage和holder进行继承以创建一个新类，它可拥有BitImage和holder。

对于MI来说，这是一个重要的理由，许多类库都是以这种模式构造的。然而，正如在第15章所看到的，模板的增加已经改变了包容器的生成方法，所以上述情况并不是 MI的有力论

点。

其他需要MI的理由是和设计相关的逻辑上的。和以上情况不同，这里有对基类的控制，同时我们为了使设计更灵活，更有用而采用 MI。这种情况可以以最初的输入输出流的类库设计为例：

输入流和输出流本身都是有用的类，但是通过对它们的继承可将两者的特性和行为加以结合而形成一个新类。

不管使用MI是出于什么样的动机，在处理过程中都会出现许多问题，必须先理解它们而后再使用它们。

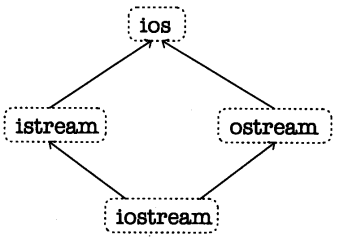


图 16-2

16.2 子对象重叠

当继承基类时，在派生类中就获得了基类所有数据成员的副本，该副本称为子对象。假若对类d1和类d2进行多重继承而形成类 mi，类mi会包含d1的子对象和d2的子对象，所以mi对象看上去如：

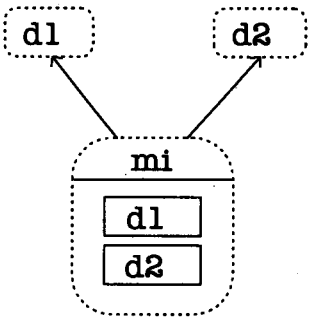


图 16-3

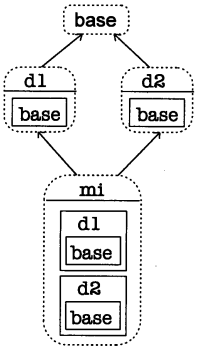


图 16-4

现在考虑如果d1和d2都是从相同基类派生的，该基类称为base，那么会发生什么呢？

在上面的图中，d1和d2都包含base的子对象，所以mi包含基的两个子对象。从继承图形状上看，有时该继承层次结构称为“菱形”。没有菱形情况时，多重继承相当简单，但是只要菱形一出现，由于新类中存在重叠的子对象，麻烦就开始了。重叠的子对象增加了存储空间，这种额外开销是否成为一个问题取决于我们的设计，但它同时又引入了二义性。

16.3 向上映射的二义性

在上图中，假若把一个指向 mi的指针映射给一个指向 base的指针时，将会发生什么呢？base有两个子对象，因此会映射成哪一个的地址呢?这里用代码来揭示上图的问题：

```
//: MULTIPL1.CPP -- MI & ambiguity
#include <iostream.h>
#include "..\14\tstash.h"

class base {
public:
  virtual char* vf() const = 0;
```

```
};

class d1 : public base {
public:
  char* vf() const { return "d1"; }
};

class d2 : public base {
public:
  char* vf() const { return "d2"; }
};

// Causes error: ambiguous override of vf():
//! class mi : public d1, public d2 {};

main() {
  tstash<base> b;
  b.add(new d1);
  b.add(new d2);
  // Cannot upcast: which subobject?:
//!  b.add(new mi);
  for(int i = 0; i < b.count(); i++)
    cout << b[i]->vf() << endl;
}
```

这里存在两个问题。首先，由于d1和d2分别对vf()定义这会导致一个冲突，所以不能生成mi类。其次，在对b[]的数组定义中试图创建一个new mi 并将类型转化为base*，由于没有办法搞清我们打算使用d1子对象的base还是d2子对象的base作为结果地址，所以编译器将不会受理。

16.4 虚基类

为了解决第一个问题，必须对类mi中的函数vf()进行重新定义以消除二义性。

对于第二个问题的解决应着眼于语言扩展，这就是对 virtual赋予新的含义。假若以virtual的方式继承一个基类，则仅仅会出现一个基类子对象。虚基类由编译器的指针法术（ pointer magic）来实现，该方法使人想起普通虚函数的实现。

由于在多重继承期间仅有一个虚基类子对象，所以在地址回溯中不会产生二义性。下面是一个例子：

```
//: MULTIPL2.CPP -- Virtual Base Classes
#include <iostream.h>
#include "..\14\tstash.h"

class base {
public:
```

```
  virtual char* vf() const = 0;
};

class d1 : virtual public base {
public:
  char* vf() const { return "d1"; }
};

class d2 : virtual public base {
public:
  char* vf() const { return "d2"; }
};
// MUST explicitly disambiguate vf():
class mi : public d1, public d2 {
public:
  char* vf() const { return d1::vf();}
};

main() {
  tstash<base> b;
  b.add(new d1);
  b.add(new d2);
  b.add(new mi); // OK
  for(int i = 0; i < b.count(); i++)
    cout << b[i]->vf() << endl;
}
```

现在编译器可以接受地址向上映射了，但是依然要对 mi中的vf()消除二义性，否则，编译器分辨不出需要使用哪一个版本。

16.4.1 “最晚辈派生”类和虚基初始化

虚基类的使用并不如此简单。以上的例子中使用的是编译器生成的缺省构造函数，如果虚基类存在一个构造函数，情况就有些不同了。为了便于理解，引入一个新术语：最晚辈派生类(most-derived)。

最晚辈派生类是当前所在的类，当考虑构造函数时它尤其重要。在前面的例子中，基构造函数里的最晚辈派生类是base；在d1构造函数中，d1是最晚辈派生类；在mi构造函数中，mi是最晚辈派生类。

打算使用一个虚基类时，最晚辈派生类的构造函数的职责是对虚基类进行初始化。这意味着该类不管离虚基类多远，都有责任对虚基类进行初始化。这里有一个初始化的例子：

```
//: MULTIPL3.CPP -- Virtual base initialization
// Virtual base classes must always be
// Intialized by the "most-derived" class
#include <iostream.h>
#include "..\14\tstash.h"
```

```
class base {
public:
  base(int) {}
  virtual char* vf() const = 0;
};

class d1 : virtual public base {
public:
  d1() : base(1) {}
  char* vf() const { return "d1"; }
};

class d2 : virtual public base {
public:
  d2() : base(2) {}
  char* vf() const { return "d2"; }
};

class mi : public d1, public d2 {
public:
  mi() : base(3) {}
  char* vf() const {
    return d1::vf(); // MUST disambiguate
  }
};

class x : public mi {
public:
  // You must ALWAYS init the virtual base:
  x() : base(4) {}
};

main() {
  tstash<base> b;
  b.add(new d1);
  b.add(new d2);
  b.add(new mi); // OK
  b.add(new x);
  for(int i = 0; i < b.count(); i++)
    cout << b[i]->vf() << endl;
}
```

d1和d2在它们的构造函数中都必须对base初始化，这和我们想象的一样。但mi和x也都如此，尽管它们和base相隔好几个层次。这是因为每一个都能成为最晚辈派生类。编译器是无法知道使用d1初始化base还是使用d2的，因而我们总是被迫在最晚辈派生类中具体指出。注意，仅仅这个被选中的虚基构造函数被调用。

16.4.2 使用缺省构造函数向虚基“警告”

为了促进最晚辈派生类初始化一个虚基类，最好通过创建一个虚基类的缺省构造函数，将虚基视为黑箱,如下面的例子。这是由于虚基可能被深深地埋藏在类层次中，它看上去繁琐而令人困惑：

```
//: MULTIPL4.CPP -- "Tying off" virtual bases
// so you don't have to worry about them
// in derived classes
#include <iostream.h>
#include "..\14\tstash.h"
class base {
public:
 // Default constructor removes responsibility:
  base(int = 0) {}
  virtual char* vf() const = 0;
};

class d1 : virtual public base {
public:
  d1() : base(1) {}
  char* vf() const { return "d1"; }
};

class d2 : virtual public base {
public:
  d2() : base(2) {}
  char* vf() const { return "d2"; }
};

class mi : public d1, public d2 {
public:
  mi() {} // Calls default constructor for base
  char* vf() const {
    return d1::vf(); // MUST disambiguate
  }
};

class x : public mi {
public:
  x() {} // Calls default constructor for base
};

main() {
  tstash<base> b;
  b.add(new d1);
```

```
  b.add(new d2);
  b.add(new mi); // OK
  b.add(new x);
  for(int i = 0; i < b.count(); i++)
    cout << b[i]->vf() << endl;
}
```

假若我们总能为虚基类安排缺省构造函数，这可使别人继承我们的类变得非常容易。

16.5 开销

术语“指针法术（pointer magic）”用于描述虚继承的实现。下面的程序可观察到虚继承的物理开销。

```
//: OVERHEAD.CPP -- Virtual base class overhead
#include <fstream.h>
ofstream out("overhead.out");

class base {
public:
  virtual void f() const {};
};

class nonvirtual_inheritance
  : public base {};

class virtual_inheritance
  : virtual public base {};

class virtual_inheritance2
  : virtual public base {};

class mi
  : public virtual_inheritance,
    public virtual_inheritance2 {};
#define WRITE(arg) \
out << #arg << " = " << arg << endl;

main() {
  base b;
  WRITE(sizeof(b));
  nonvirtual_inheritance nonv_inheritance;
  WRITE(sizeof(nonv_inheritance));
  virtual_inheritance v_inheritance;
  WRITE(sizeof(v_inheritance));
  mi MI;
  WRITE(sizeof(MI));
}
```

这些都包含一个单字节，该字节是“内核长度（core size）”。由于所有类都包含虚函数，由于有一个指针，因此对象长度会比内核长度大（起码编译器为了校正定位会把额外的字节添入对象中），然而结果却有点让人吃惊。（该结果来自特定的编译器，不同版本可能有所不同。）

```
sizeof(b)=2
sizeof(nonv_inheritance)=2
sizeof (v_inheritance)=6
sizeof(MI)=12
```

正如所料，b和nonv_inheritance包含有额外指针。但当虚继承时，显示出VPTR和两个额外指针被加了进去。到了多重继承执行时，对象显示出它拥有五个额外指针（然而，指针之一可能是针对第二个多重继承子对象的VPTR。）

这种奇怪的情况可以通过探究我们的特殊实现和考查成员选择的汇编语言，以准确地确定这些额外字节是为什么而设计的以及用多重继承成员选择的开销有多大[1]。总之，虚拟多重继承不过是权益之计，在重视效率的情况下，应该保守地（或避免）使用它。

16.6 向上映射

无论是通过创建成员对象还是通过继承的方式，当我们把一个类的子对象嵌入一个新类中时，编译器会把每一个子对象置于新对象中。当然，每一个子对象都有自己的 this指针，在处理成员对象的时候可以万事俱简。但是只要引入多重继承，一个有趣的现象就会出现：由于对象在向上映射期间出现多个类，因而对象存在多个this指针。下面就是这种情况的例子：

```
//: MITHIS.CPP -- MI and the "this" pointer
#include <fstream.h>
ofstream out("mithis.out");

class base1 {
  char c[0x10];
public:
  void printthis1() {
    out << "base1 this = " << this << endl;
  }
};

class base2 {
  char c[0x10];
public:
  void printthis2() {
    out << "base2 this = " << this << endl;
  }
};

class member1 {
  char c[0x10];
public:
```

[1] 看Jan Gray C++Under the Hood, a chapter in Black Belt C++(edited by Bruce Eckel, M& T press, 1995)。

```
  void printthism1() {
    out << "member1 this = " << this << endl;
  }
};

class member2 {
  char c[0x10];
public:
  void printthism2() {
    out << "member2 this = " << this << endl;
  }
};

class mi : public base1, public base2 {
  member1 m1;
  member2 m2;
public:
  void printthis() {
    out << "mi this = " << this << endl;
    printthis1();
    printthis2();
    m1.printthism1();
    m2.printthism2();
  }
};

main() {
  mi MI;
  out << "sizeof(mi) = "
    << hex << sizeof(mi) << " hex" << endl;
  MI.printthis();
  // A second demonstration:
  base1* b1 = &MI; // Upcast
  base2* b2 = &MI; // Upcast
  out << "base 1 pointer = " << b1 << endl;
  out << "base 2 pointer = " << b2 << endl;
}
```

例子中由于每个类的数组字节数都是用十六进制长度来创建的，所以用十六进制数打出的输出地址是容易阅读的。每个类都有一个打印 this指针的函数，这些类通过多重继承和组合而被装配成类mi，它打印自己和其他所有子对象的地址，由主程序调用这些打印功能。可以清楚地看到，能在一个相同的对象中获得两个不同的 this指针。下面是mi及向上映射到两个不同类的地址输出：

```
sizeof(mi)=40 hex
mi this=0x223e
base1 this=0x223e
```

```
base2 this=0x224e
member1 this=0x225e
member2 this=0x226e
base 1 pointer=0x223e
base 2 pointer=0x224e
```

虽然上述输出根据编译器的不同而有所不同，并且在标准 C++中亦未做详细说明，但它仍具有相当的典型性。派生对象的起始地址和它的基类列表中的第一个类的地址是一致的，第二个基类的地址随后，接着根据声明的次序安排成员对象的地址。

当向base1和base2进行映射时，产生的指针表面上是指向同一个对象，而实际上则必然有不同 this指针，只有这样，固有的起始地址能够被传给相应子对象的成员函数。假若当我们为多重继承的子对象调用一个成员函数时，隐式向上映射就会产生，而只有上述方法才能使程序正确工作。

持久性

由于打算调用与多重继承对象的子对象相关的成员函数，持久性通常并不是一个问题。然而，假若成员函数需要知晓对象的真实起始地址，多重继承就会出现问题。反而言之，多重继承有用的一种情况是拥有持久性。

局部对象的生命周期由其定义确定范围，全局对象的生命周期就是程序的生命周期，持久对象则存在于程序请求之间，通常它被认为存在于磁盘上而非存储器中。面向对象数据库的定义就是“ 一个持久对象集 ”。

为了实现持久性，必须把一个持久对象从磁盘中移入存储器以便为其调用函数，稍后在程序任务完成前还必须把它存入磁盘。把对象存入磁盘涉及四个方面的内容：

1) 必须把对象在存储器中的表示转化成磁盘上的字节序列。

2) 由于存储器中的指针值在下一次程序启用时已毫无意义，所以必须把它们转化成有意义的东西。

3) 指针指向的东西也必须被存储和取回。

4) 当从磁盘到存储器上重组一个对象时，必须考虑对象中的虚指针。

将对象从存储器中转换到磁盘上的过程（把对象写入磁盘）称为串行化（ serialization）；而把对象恢复重组到存储器中的过程称为反串行化（ deserialization）。尽管这些过程十分方便，但由于处理的开销过大而使语言不能直接支持它。类库常常根据增加特定的成员函数以及在新类上设置该需求而支持串行化和反串行化。（通常每个新类都有特定的serialize() 函数）持久性的实现一般不是自动完成的，通常必须对对象进行显式读写。

1. 基于MI的持久性

现在可以考虑跨越指针问题，考虑创建一个使用多重继承把持久性装入简单对象的类。通过继承这个persisence类连同我们的新类，我们可以自动地创建能读写磁盘的类。这听起来很好，但是使用多重继承会引入下例所示的一个缺陷。

```
//: PERSIST1.CPP -- Simple persistence with MI
#include <fstream.h>

class persistent {
  int objSize; // Size of stored object
public:
```

```
  persistent(int sz) : objSize(sz) {}
  void write(ostream& out) const {
    out.write((char*)this, objSize);
  }
  void read(istream& in) {
    in.read((char*)this, objSize);
  }
};
class data {
  float f[3];
public:
  data(float f0 = 0.0, float f1 = 0.0,
    float f2 = 0.0) {
    f[0] = f0;
    f[1] = f1;
    f[2] = f2;
  }
  void print(const char* msg = "") const {
    if(*msg) cout << msg << "   ";
    for(int i = 0; i < 3; i++)
      cout << "f[" << i << "] = "
           << f[i] << endl;
  }
};

class wdata1 : public persistent, public data {
public:
  wdata1(float f0 = 0.0, float f1 = 0.0,
    float f2 = 0.0) : data(f0, f1, f2),
    persistent(sizeof(wdata1)) {}
};

class wdata2 : public data, public persistent {
public:
  wdata2(float f0 = 0.0, float f1 = 0.0,
    float f2 = 0.0) : data(f0, f1, f2),
    persistent(sizeof(wdata2)) {}
};

main() {
  {
    ofstream f1("f1.dat"), f2("f2.dat");
    wdata1 d1(1.1, 2.2, 3.3);
    wdata2 d2(4.4, 5.5, 6.6);
    d1.print("d1 before storage");
```

```
    d2.print("d2 before storage");
    d1.write(f1);
    d2.write(f2);
  } // Closes files
  ifstream f1("f1.dat"), f2("f2.dat");
  wdata1 d1;
  wdata2 d2;
  d1.read(f1);
  d2.read(f2);
  d1.print("d1 after storage");
  d2.print("d2 after storage");
}
```

在上面的简单版本中，persistent::read() 和persistent::write()函数可获取this指针并调用输入输出流的read() 和 write() 函数（注意，任意类型的IO流都可使用）。一个更为复杂的持久性类可能为每个子对象安排虚write()函数。

到目前为止，本书涉及的语言特性尚不能使持久性类知道对象的字节长度，所以在构造函数中插入了一个长度参数。（在第18章中，“运行时类型识别”会揭示怎样找到仅由一个基指针所给定对象的确切类型，一旦获得确切类型，就可以使用 sizeof运算符而求得正确的长度。）

类data不含有指针或VPTR，所以向磁盘写及从磁盘读运算是安全的。类 wdata1在main()中向F1.DAT写入，稍后把数据从文件中取出，没有什么问题。然而当把 persistent置于类wdata2的继承列表中的第二项时，persistent的this指针指向对象的末端，所以读写运算的内容会超出对象的尾部。这样从磁盘文件中读取的内容毫无价值而且会对安排在该对象之后的存储内容造成破坏。

在多重继承中，这种问题是在类必需从子对象的this指针产生实际对象的this指针时发生的。当然，我们知道编辑器是根据继承列表中的类声明次序而安排对象，所以应把重要的类放置在列表的首部（假定只有一个重要类），然而该类可能存在于其他类的继承层次中，这可能会使我们无意识地把该类置于错误的位置上。幸运的是，即使使用了多重继承，在第 18章中介绍的“运行时类型识别”技术也会产生指向实际对象的正确指针。

2. 改良的持久性

下面是一个更具实践化、经常使用的持久性方法的例子。该例子在基类中创建了具有读写功能的虚函数，当类不断派生时，它要求每个新类的创建者能重载这些虚函数。函数的参数是可读出或写入的流对象 [1]。作为类的创建者，我们知道怎样对新的部分进行读写，有责任创建正确的函数调用。该例子没有前面例子的精巧的质量，它要求部分用户有更多的知识和参加更多的编码工作，但该方法并不会由于提交指针而出错。

```
//: PERSIST2.CPP -- Improved MI persistence
#include <fstream.h>
#include <string.h>

class persistent {
public:
```

[1] 有时流只有一个函数，参数包括打算读还是写的信息。

```
  virtual void write(ostream& out) const = 0;
  virtual void read(istream& in) = 0;
};
class data {
protected:
  float f[3];
public:
  data(float f0 = 0.0, float f1 = 0.0,
    float f2 = 0.0) {
    f[0] = f0;
    f[1] = f1;
    f[2] = f2;
  }
  void print(const char* msg = "") const {
    if(*msg) cout << msg << endl;
    for(int i = 0; i < 3; i++)
      cout << "f[" << i << "] = "
           << f[i] << endl;
  }
};

class wdata1 : public persistent, public data {
public:
  wdata1(float f0 = 0.0, float f1 = 0.0,
    float f2 = 0.0) : data(f0, f1, f2) {}
  void write(ostream& out) const {
    out << f[0] << " " << f[1] << " " << f[2];
  }
  void read(istream& in) {
    in >> f[0] >> f[1] >> f[2];
  }
};

class wdata2 : public data, public persistent {
public:
  wdata2(float f0 = 0.0, float f1 = 0.0,
    float f2 = 0.0) : data(f0, f1, f2) {}
  void write(ostream& out) const {
    out << f[0] << " " << f[1] << " " << f[2];
  }
  void read(istream& in) {
    in >> f[0] >> f[1] >> f[2];
  }
};

class conglomerate : public data,
```

```
  public persistent {
    char* name; // Contains a pointer
    wdata1 d1;
    wdata2 d2;
  public:
    conglomerate(const char* nm = "",
      float f0 = 0.0, float f1 = 0.0,
      float f2 = 0.0, float f3 = 0.0,
      float f4 = 0.0, float f5 = 0.0,
      float f6 = 0.0, float f7 = 0.0,
      float f8= 0.0) : data(f0, f1, f2),
      d1(f3, f4, f5), d2(f6, f7, f8) {
      name = new char[strlen(nm) + 1];
      strcpy(name, nm);
    }
    void write(ostream& out) const {
      int i = strlen(name) + 1;
      out << i << " "; // Store size of string
      out << name << endl;
      d1.write(out);
      d2.write(out);
      out << f[0] << " " << f[1] << " " << f[2];
    }
    // Must read in reverse order as write:
    void read(istream& in) {
      delete []name; // Remove old storage
      int i;
      in >> i >> ws; // Get int, strip whitespace
      name = new char[i];
      in.getline(name, i);
    d1.read(in);
    d2.read(in);
    in >> f[0] >> f[1] >> f[2];
  }
  void print() const {
    data::print(name);
    d1.print();
    d2.print();
  }
};

main() {
  {
    ofstream data("data.dat");
    conglomerate C("This is conglomerate C",
      1.1, 2.2, 3.3, 4.4, 5.5,
```

```
      6.6, 7.7, 8.8, 9.9);
    cout << "C before storage" << endl;
    C.print();
    C.write(data);
  } // Closes file
  ifstream data("data.dat");
  conglomerate C;
  C.read(data);
  cout << "after storage: " << endl;
  C.print();
}
```

基类persistent的纯虚函数在派生类中必须被重载以执行正确的读写运算。假若我们已经知道data是持久的，可以直接继承它并在那里重载虚函数，因而可不必使用多重继承。本例的思想是在于我们不拥有data的代码，这些代码在别处已经创建了，它可能是其他类层次的一部分（我们不能控制它的继承）。然而，为了使这个方案能正确地工作，我们必须对下层实现能访问，使得它能被存放，所以我们使用了protected。

类wdata1和wdata2使用了常见的IO流插入器和取出器以便向流对象存储和从流对象取回data的保护性数据。在write()中，我们可以看到每个浮点数后都加了一个空格，这对读入数据的分解是必要的。类conglomerate不仅继承了data，而且拥有两个wdata1和wdata2类型的成员对象及一个字符串指针。另外，由persistent派生的所有类也包含一个VPTR，所以该例子揭示了使用持久性所遇到的一些问题。

当创建write()和read()函数对时，read()函数必须对发生在write()期间的东西进行准确镜像，所以可通过read()抽取磁盘上由write()安置的比特流。这里的第一个问题是char*，它指向一个任意长的字符串。对字符串进行计算并在磁盘上存储其长度，这可使read()函数能正确地分配存储容量。

当拥有的子对象具有read()和write()成员函数时，我们所需要做的是在新的函数read()和write()中调用基类的成员函数。

它的后面跟着基类直接存储的成员函数。

人们在自动持久性方面已竭尽全力。例如，定义类时创建了修改过的预处理器以支持“持久性”主题。你可以想出一个实现持久性更好的方法，但上述方法的优点是它在C++实现下工作，无需特别的语言扩展，相对较健壮。

16.7 避免MI

在PERSIST2.CPP中对多重继承的使用有一点人为的因素，它考虑到在项目中一些类代码不受程序员控制的情况。对以上的例子进行细查，可以看到，通过使用data类型的成员对象以及把虚read()和write()成员放入data或wdata1和wdata2中而不是置于一个独立的类中，这样MI是可以避免使用的。语言会包含一些不常用的特性，这种特殊性只有在其他方法困难或者不可能处理时才使用。当出现是否使用多重继承的问题时，我们可以先问自己两个问题：

1) 我们有必要同时使用两个类的公共接口吗，是否可在一个类中用成员函数包含这些接口呢？

2) 我们需要向上映射到两个基类上吗？（当然，在我们有两个以上的基类被应用。）

如果我们对以上两个问题都能以“不”来回答，那么就可以避免使用 MI。

需要注意，当类仅仅需要作为一个成员参数被向上回溯的情况。在这种情况下，该类可以被嵌入，同时可由新类的自动类型转化运算符产生一个针对被嵌入对象的引用。当将新类的对象作为参数传给希望以嵌入对象为参数的函数时，都将发生类型转换。然而类型转换不能用于通常的成员选择，这时需要继承。

16.8 修复接口

使用多重继承的最好的理由之一是使用控制之外的代码。假定已经拥有了一个由头文件和已经编译的成员函数组成的库，但是没有成员函数的源代码。该库是具有虚函数的类层次，它含有一些使用库中基类指针的全局函数，这就是说它多态地使用库对象。现在，假定我们围绕着该库创建了一个应用程序并且利用基类的多态方式编写了自己的代码。

在随后的项目开发或维护期间，我们发现基类接口和供应商所提供的不兼容：我们所需要的是某虚函数而可能提供的却是非虚的，或者对于解决我们的问题的基本虚函数在接口中根本不存在。假若有源代码则可以返回去修改，但是我们没有，我们有大量的依赖最初接口的已生成的代码，这时多重继承则是极好的解决方法。

下面的例子是所获得的库的头文件：

```
//: VENDOR.H -- Vendor-suppplied class header
// You only get this & the compiled VENDOR.OBJ
#ifndef VENDOR_H_
#define VENDOR_H_

class vendor {
public:
  virtual void v() const;
  void f() const;
  ~vendor();
};

class vendor1 : public vendor {
public:
  void v() const;
  void f() const;
  ~vendor1();
};

void A(const vendor&);
void B(const vendor&);
// Etc.
#endif // VENDOR_H_
```

假定库很大并且有更多的派生类和更大的接口。注意，它包含函数 A()和B()，以基类指针为参数。下面是库的实现文件：

```
//: VENDOR.CPP -- Implementation of VENDOR.H
// This is compiled and unavailable to you
```

```
#include <fstream.h>
#include "..\15\vendor.h"

extern ofstream out; // For trace info

void vendor::v() const {
  out << "vendor::v()\n";
}

void vendor::f() const {
  out << "vendor::f()\n";
}

vendor::~vendor() {
  out << "~vendor()\n";
}

void vendor1::v() const {
  out << "vendor1::v()\n";
}
void vendor1::f() const {
  out << "vendor1::f()\n";
}

vendor1::~vendor1() {
  out << "~vendor1()\n";
}

void A(const vendor& V) {
  // ...
  V.v();
  V.f();
  //...
}

void B(const vendor& V) {
  // ...
  V.v();
  V.f();
  //...
}
```

在我们的项目中，这些源代码是不能得到的，而我们得到的是已编译过的 VENDOR.OBJ或VENDOR.LIB文件（或系统中相应的等价物）。

使用该库会产生问题。首先析构函数不是虚的，这实际上是创建者的一个设计错误。另外，

f()也不是虚的，这可能是库的创建者认为没有必要。但是我们会发现基类接口失去了解决前述问题的必要能力。假若我们已经使用已存在的接口（不包含函数A()和B()，因为它们不受控制）编制了大量代码，而且并不打算改变它。

为了补救该问题，我们可以创建自己的类接口以及从我们的接口和已存在的类中进行多重继承，以便生成一批新类：

```
//: PASTE.CPP -- Fixing a mess with MI
#include <fstream.h>
#include "..\15\vendor.h"

ofstream out("paste.out");

class mybase { // Repair vendor interface
public:
  virtual void v() const = 0;
  virtual void f() const = 0;
  // New interface function:
  virtual void g() const = 0;
  virtual ~mybase() { out << "~mybase()\n"; }
};

class paste1 : public mybase, public vendor1 {
public:
  void v() const {
    out << "paste1::v()\n";
    vendor1::v();
  }
  void f() const {
    out << "paste1::f()\n";
    vendor1::f();
  }
  void g() const {
    out << "paste1::g()\n";
  }
  ~paste1() { out << "~paste1()\n"; }
};

main() {
  paste1& p1p = *new paste1;
mybase& mp = p1p; // Upcast
out << "calling f()\n";
mp.f();  // Right behavior
out << "calling g()\n";
mp.g(); // New behavior
out << "calling A(p1p)\n";
A(p1p); // Same old behavior
```

```
  out << "calling B(p1p)\n";
  B(p1p);  // Same old behavior
  out << "delete mp\n";
  // Deleting a reference to a heap object:
  delete &mp; // Right behavior
}
```

在mybase(它不使用MI)中 ,f()和析构函数都改成虚的，并在接口中增加了新的虚函数 g()。现在，每一个原来的派生类都必须重新创建，采用 MI使其搀入到一个新接口中。函数paste1::v() 和paste1::f()仅需要调用该函数原先基类的版本。但是，如果现在在 main()中对mybase进行向上映射：

```
mybase * mp=plp;// upcast
```

这样，所有函数的调用包括delete都通过多态的mp来完成，同样对新接口函数g()的调用也通过mp。下面是程序的输出：

```
calling f()
paste1::f()
vendor1::f()
calling g()
paste1::g()
calling A(p1p)
paste1::v()
vendor1::v()
vendor::f()
calling B(p1p)
paste1::v()
vendor1::v()
vendor::f()
delete mp
~paste1()
~vendor1()
~vendor()
~mybase()
```

原先的库函数A()和B()仍然可以工作（若新的v()调用它的基类版本）。析构函数现在是虚的并表现了正确的行为。

虽然这是一个散乱的例子，但它确实可以在实际中出现，同时很好地说明了多重继承在何处是必要的：我们必须能够向上映射到两个基类。

16.9 小结

除C++之外，其他OOP语言都不支持多重继承，这是由于针对OOP来说C++是一个“混血”版，它不能像Smalltalk那样把类强制安排成一个单一完整的类层次。 C++支持许多不同形式的继承树，有时需要结合两个或更多的继承树接口形成一个新类。

假若在类层次中没有出现“菱形”形状， MI是相当简单的，尽管必须解决在基类中相同的函数标识。假若出现“菱形”形状，我们必须处理由于引入虚基类导致的子对象重叠问题。

这不仅增加了混乱而且使底层变得更为复杂和缺乏效率。多重继承被 Zack Urlocker称为“九十年代的goto”，因为它确实像goto。在通常编程开发时，应避免使用多重继承，只是在某一时候它才变得非常有用。MI是C++中“次要的”但更为高级的特性，它被设计用于处理特定的情况。如果我们发现经常使用它，应该调查一下使用它的原因，应基于 Occam所提出的简单完美性原则（Occam’s Razor）“我必须向上映射到所有基类吗”，如果我们的回答是否定的，则用嵌入所有基类实例的方法会更容易，而不必使用向上映射。

16.10 练习

1. 本练习会使我们一步一步地穿过 MI陷阱。创建一个含有单个 int参数的构造函数和返回为void型的无参数成员函数f()的基类X。从X派生出Y和Z，为Y和Z各创建一个单个int参数的构造函数。通过多重继承从Y和Z中派生出A。生成一个类A的对象并为对象调用f()。以明显无二义性的方式解决这个问题。

2. 创建一个指向X的指针px，将类型A的对象的地址在它被创建前赋予 px。注意虚基类的使用问题。现在修改X，使得这时不必再在A中为X调用构造函数。

3. 移去f()的明显无二义性说明，观察能否通过px调用f()。对其跟踪以便观察哪个函数被调用。注意这个问题以在一个类层次中调用正确的函数。

第17章　异 常 处 理

错误修复技术的改进是提高代码健壮性的最有效方法之一。

但是,大多数程序设计人员在实际设计中往往忽略出错处理 ,似乎是在没有错误的状态下编程。毫无疑问，出错处理的繁琐及错误检查引起的代码膨胀是导致上述问题的主要原因。例如，虽然printf( )函数可返回打印参数的个数，但是实际程序设计中没有人检查该值。出错处理引起的代码膨胀将不可避免地增加程序阅读的困难，这对于程序设计人员来说是十分令人烦恼的。

C语言中实现出错处理的方法是将用户函数与出错处理程序紧密地结合起来，但是这将造成出错处理使用的不方便和难以接受。

异常处理是C++语言的一个主要特征，它提出了出错处理更加完美的方法。

1) 出错处理程序的编写不再繁琐，也不须将出错处理程序与“ 通常 ”代码紧密结合。在错误有可能出现处写一些代码，并在后面的单独节中加入出错处理程序。如果程序中多次调用一个函数，在程序中加入一个函数出错处理程序即可。

2) 错误发生是不会被忽略的。如果被调用函数需发送一条出错信息给调用函数，它可向调用函数发送一描述出错信息的对象。如果调用函数没有捕捉和处理该错误信号，在后续时刻该调用函数将继续发送描述该出错信息的对象，直到该出错信息被捕捉和处理。

在这一章中我们将讨论C语言的出错处理方法，讨论为何该方法在C语言中不是很理想的，并且无法在C++中使用；然后学习try，throw和catch的用法，它们在C++中支持异常处理。

17.1　C语言的出错处理

本书在第8章以前使用C标准库的assert( )宏作为出错处理的方法。第8章以后assert( )被按照原先的设计目的使用：在开发过程中，使用它们，完成后用 #define NDEBUG使之失效，以便推出产品。为了在运行时检查错误， assert( )被allege( )函数和第8章中引入的宏所取代。通常我们会说：“ 对于出错处理我们必须面对复杂的代码，但是在这个例子中我们不必由此感到烦恼 ”。allege( )函数对一些小型程序很方便，对于复杂的大型程序，所编写的出错处理程序也将更加复杂。

在通过检查条件我们能确切地知道做什么的情况下，出错处理就变得十分明确和容易了，因为我们通过上下文得到了所有必要的信息。当然，我们只是在这一点上处理错误。这些都是十分普通的错误，不是这一章的主题。

若错误问题发生时在一定的上下文环境中得不到足够的信息，则需要从更大的上下文环境中提取出错处理信息，下面给出了C语言处理这类情况的三种典型方法。

1) 出错信息可通过函数的返回值获得。如果函数返回值不能用，则可设置一全局错误判断标志（标准C语言中errno( )和perror( )函数支持这一方法）。正如前文提到的，由于对每个函数调用都进行错误检查，这十分繁琐并增加了程序的混乱度。程序设计者可能简单地忽略这些出错信息，因为乏味而迷乱的错误检查必须随着每个函数调用而出现。另外，来自偶然出现异常的函数的返回值可能并不反映什么问题。

2) 可使用C标准库中一般不太熟悉的信号处理系统，利用signal()函数（判断事件发生的类型）和raise()函数（产生事件）。由于信号产生库的使用者必须理解和安装合适的信号处理系统，所以应紧密结合各信号产生库，但对于大型项目，不同库之间的信号可能会产生冲突。

3) 使用C标准库中非局部的跳转函数：setjmp( ) 和 longjmp( )。setjmp( ) 函数可在程序中存储一典型的正常状态，如果进入错误状态， longjmp( )可恢复setjmp( ) 函数的设定状态，并且状态被恢复时的存储地点与错误的发生地点紧密联系。

考虑C++语言的出错处理方案时会存在另一个关键性问题：由于 C语言的信号处理技术和setjmp/longjmp函数不能调用析构函数，所以对象不能被正确地清除。由于对象不能被清除，它将被保留下来并且将不能再次被存取，所以存在这种问题时实际上是不可能有效正确地从异常情况中恢复出来。下面的例子将演示setjmp/longjmp的这一特点：

```
//: NONLOCAL.CPP -- setjmp & longjmp
#include <iostream.h>
#include <setjmp.h>

class rainbow {
public:
  rainbow() { cout << "rainbow()" << endl; }
  ~rainbow() { cout << "~rainbow()" << endl; }
};

jmp_buf kansas;

void OZ() {
  rainbow RB;
  for(int i = 0; i < 3; i++)
    cout << "there's no place like home\n";
  longjmp(kansas, 47);
}

main() {
  if(setjmp(kansas) == 0) {
    cout << "tornado, witch, munchkins...\n";
    OZ();
  } else {
    cout << "Auntie Em! "
         << "I had the strangest dream..."
         << endl;
  }
}
```

setjmp()是一个特别的函数，因为如果我们直接调用它，它就把当前进程状态的所有相关信息存放在jmp_buf中，并返回零。这样，它的行为象通常的函数。然而，如果使用同一个jmp_buf调用longjmp()，这就象再次从setjmp()返回，即正确地弹出setjmp()的后端。这时，返回值对于longjmp()是第二个参数，所以能发现实际上从longjmp()中返回了。可以想象，有多个

不同的jmp_buf，可以弹出程序的多个不同位置的信息。局部 goto（用标号）和这个非局部跳转的不同在于我们能通过setjmp/longjmp跳转到任何地方（一些限制不在这里讨论）。

在C++中的问题是，longjmp()不适用于对象，特别是，当它跳出范围时它不调用析构函数[1]。析构函数调用是必须的，所以这种方法在C++中不可行。

17.2 抛出异常

如果程序发生异常情况，而在当前的上下文环境中获取不到异常处理的足够信息，我们可以创建一包含出错信息的对象并将该对象抛出当前上下文环境，将错误信息发送到更大的上下文环境中。这称为异常抛出。如：

```
throw myerror ("something bad happened");
```

myerror是一个普通类，它以字符变量作为其参数。当进行异常抛出时我们可使用任意类型变量作为其参数（包括内部类型变量），但更为常用的办法是创建一个新类用于异常抛出。

关键字throw的引入引起了一系列重要的相关事件发生。首先是 throw调用构造函数创建一个原执行程序中并不存在的对象。其次，实际上这个对象正是 throw函数的返回值，即使这个对象的类型不是函数设计的正常返回类型。对于交替返回机制，如果类推太多有可能会陷入困境，但仍可看作是异常处理的一种简单方法，可通过抛出一个异常来退出普通作用域并返回一个值。

因为异常抛出同常规函数调用的返回地点完全不同，所以返回值同普通函数调用具有很小的相似性（异常处理器地点与异常抛出地点可能相差很远）。另外，只有在异常时刻成功创建的对象才被清除掉。（常规函数调用则不同，它使作用域内的所有对象均被清除。）当然，异常情况产生的对象本身在适当的地点也被清除。

另外，我们可根据要求抛出许多不同类型的对象。一般情况下，对于每种不同的错误可设定抛出不同类型的对象。采用这样的方法是为了存储对象中的信息和对象的类型，所以别人可以在更大的上下文环境中考虑如何处理我们的异常。

17.3 异常捕获

如果一个函数抛出一个异常，它必须假定该异常能被捕获和处理。正如前文所提到的，允许对一个问题集中在一处解决，然后处理在别处的差错，这也正是 C++语言异常处理的一个优点。

17.3.1 try块

如果在函数内抛出一个异常（或在函数调用时抛出一个异常），将在异常抛出时退出函数。如果不想在异常抛出时退出函数，可在函数内创建一个特殊块用于解决实际程序中的问题（和潜在产生的差错）。由于可通过它测试各种函数的调用，所以被称为测试块。测试块为普通作用域，由关键字try引导：

```
try {
    // code that may generate exceptions
}
```

[1] 当我们运行这个例子时会惊奇地发现 —— 一些C++编译器调用 longjmp( )函数清除堆栈中的对象。这是兼容性差的问题。

如果没有使用异常处理而是通过差错检查来探测错误，即使多次调用同一个函数，也不得不围绕每个调用函数重复进行设置和代码检测。而使用异常处理时不需做差错检查，可将所有的工作放入测试块中。这意味着程序不会由于差错检查的引入而变得混乱，从而使得程序更加容易编写，其可读性也大为改善。

17.3.2 异常处理器

异常抛出信号发出后，一旦被异常器处理接收到就被销毁。异常处理器应具备接受任何一种类型的异常的能力。异常处理器紧随try块之后，处理的方法由关键字catch引导。

```
try {
// code that may generate exceptions
} catch(type1 id1) {
  // handle exceptions of type1
} catch(type2 id2) {
  // handle exceptions of type2
}
// etc...
```

每一个catch语句（在异常处理器中）就相当于一个以特殊类型作为单一参数的小型函数。异常处理器中标识符（id1、id2 等）就如同函数中的一个参数。如果异常抛出给出的异常类型足以判断如何进行异常处理，则异常处理器中的标识符可省略。

异常处理部分必须直接放在测试块之后。如果一个异常信号被抛出，异常处理器中第一个参数与异常抛出对象相匹配的函数将捕获该异常信号，然后进入相应的 catch语句，执行异常处理程序。catch语句与switch语句不同，它不需要在每个case语句后加入break用以中断后面程序的执行。

注意，在测试块中不同的函数的调用可能会产生相同的异常情况，但是，这时只需要一个异常处理器。

• 终止与恢复

在异常处理原理中含有两个基本模式：终止与恢复。假设差错是致命性的，当异常发生后将无法返回原程序的正常运行部分，这时必须调用终止模式（ C++支持）结束异常状态。无论程序的哪个部分只要发生异常抛出，就表明程序运行进入了无法挽救的困境，应结束运行的非正常状态，而不应返回异常抛出之处。

另一个为恢复部分。恢复意味着希望异常处理器能够修改状态，然后再次对错误函数进行检测，使之在第二次调用时能够成功运行。如果要求程序具有恢复功能，就希望程序在异常处理后仍能继续正常执行程序，这样，异常处理就更象一个函数调用 —— C ++程序中在需要进行恢复的地方如何设置状态（换言之就是使用函数调用，而非异常抛出来解决问题）。另外也可将测试块放入while循环中，以便始终装入测试块直到恢复成功得到满意的结果。

过去，程序员们使用的支持恢复性异常处理的操作系统最终被终止性模式所取代，它取消了恢复性模式。所以虽然恢复性模式初听起来是十分吸引人的，但在实际运用中却并非十分有效。其中一个原因可能是异常发生与异常处理相距较远的缘故。要终止相距较远的异常处理器，但是由于异常可能由很多地点产生，所以对于一个大型系统，从异常处跳转到异常处理器再跳转返回，这在概念上是十分困难的。

17.3.3 异常规格说明

可以不向函数使用者给出所有可能抛出的异常，但是这一般被认为是非常不友好的，因为这意味着他无法知道该如何编写程序来捕获所有潜在的异常情况。当然，如果他有源程序，他可寻找异常抛出的说明，但是库通常不以源代码方式提供。C++语言提供了异常规格说明语法，我们以可利用它清晰地告诉使用者函数抛出的异常的类型，这样使用者就可方便地进行异常处理。这就是异常规格说明，它存在于函数说明中，位于参数列表之后。

异常规格说明再次使用了关键字 throw，函数的所有潜在异常类型均随着关键字 throw而插入函数说明中。所以函数说明可以带有异常说明如下：

```
void  f ( )  throw ( toobig, toosmall, divzero) ;
```

而传统函数声明：

```
void  f ( );
```

意味着函数可能抛出任何一种异常。

如果是：

```
void  f ( )  throw ( );
```

这意味着函数不会有异常抛出。

为了得到好的程序方案和文件，为了方便函数调用者，每当写一个有异常抛出的函数时都应当加入异常规格说明。

1. unexpected( )

如果函数实际抛出的异常类型与我们的异常规格说明不一致，将会产生什么样的结果呢？这时会调用特殊函数unexpected( )。

2. set_unexpected( )

unexpected( )是使用指向函数的指针而实现的，所以我们可通过改变指针的指向地址来改变相对应的运算。这些可通过类似于 set_new_handler( )的函数set_unexpected( )来实现，set_unexpected( ) 函数可获取不带输入和输出参数的函数地址和 void返回值。它还返回unexpected指针的先前值，这样我们可存储unexpected( )函数的原先指针值，并在后面恢复它。为了使用set_unexpected( )函数，我们必须包含头文件EXCEPT.H。下面给出一实例展示本章所讨论的各个特点的简单使用：

```
//: EXCEPT.CPP -- Basic exceptions
// Exception specifications & unexpected()
#include <except.h>
#include <iostream.h>
#include <stdlib.h>
#include <string.h>

class up {};
class fit {};
void g();

void f(int i) throw (up, fit) {
  switch(i) {
    case 1: throw up();
    case 2: throw fit();
```

```
  }
  g();
}

// void g() {} // version 1
void g() { throw 47; } // Version 2
// (can throw built-in types)

void my_unexpected() {
  cout << "unexpected exception thrown";
  exit(1);
}

main() {
  set_unexpected(my_unexpected);
  // (ignores return value)
  for(int i = 1; i <=3; i++)
    try {
      f(i);
    } catch(up) {
      cout << "up caught" << endl;
    } catch(fit) {
      cout << "fit caught" << endl;
    }
}
```

作为异常抛出类，up 和fit分别被创建。通常异常类均是小型的，但有时它们包含许多额外信息，这样异常处理器可通过查询它们来获得辅助信息。

f( )函数在它的异常规格说明中声明函数的异常抛出只能是类 up 和 fit，并且函数体的定义同函数的异常规格说明是一致的。函数 g( ) （version1）被函数f( )调用，但并不抛出异常，因此这也是可行的。当函数g( ) （version1）被修改以后得g( ) （version2），g( ) （version2）仍是f( )的调用函数，但其具有异常抛出功能。函数g( )修改以后f( )函数具有了新的异常抛出，但最初创建的f( )函数对于这些却未加声明，这样就违反了异常规格说明。

my_unexpected( )函数可以没有输入或输出参数， 它是按照定制的unexpected( )函数的正确格式编写的。它仅仅打出一条有关异常的信息就退出，所以一旦被调用，我们就可以观察到这条信息。新函数unexpected( )不必有返回值（可以按照这种方法编写程序，但这是错误的）。然而它却可抛出另一个异常（也可使它抛出同一个异常），或者调用函数exit( )或abort( )。如果函数unexpected( )抛出一个异常，异常处理器将在异常抛出时开始搜寻 unexcepted异常。（这种特点对于unexcepted （）来说是独特的）

虽然new_handler( )函数的指针可为空，但unexpected( )函数的指针却不能为空。它的缺省值指向terminate( )（后面将会介绍）函数，但是，只要我们使用异常抛出和异常规格说明，我们就应该编写自己的unexpected( )函数，用于记录或者再次抛出异常及抛出新的异常或终止程序运行。

在主程序中，为了对所有的潜在异常进行检测，测试块被放入 for循环中。注意这里提到的

实现方法很象前文介绍的恢复模式，将测试块放入 for, while, do 或 if 的循环语句中，并利用每一个异常来试图消除差错问题；然后再一次的调用测试块对潜在异常进行检测。

由于程序中f( ) 的函数声明引入了up和fit两类异常，因此只有该两类异常可被抛出。因为f( ) 的函数声以后要抛出的整型，所以修改后的 g( )（version2）会使得函数my_unexpected( ) 被调用。（我们可使用任意的异常类型，包括内部类型。）

函数set_unexcepted( )被调用后，它的返回值可被忽略，但也可以被保存为函数指针，并在随后用于恢复 unexcepted( )的原先指针。

17.3.4 更好的异常规格说明

我们可能觉得在前面介绍的已存在的异常规格说明规则并非十分可靠，并且

```
void f ( ) ;
```

应该意味着函数没有异常抛出，但按照前面的规则这正好相反，它表示可抛出任意类型的异常。如果程序员要抛出任意类型的异常，我们可能会想他应该说明如下

```
void  f ( )  throw (. . .) ;  //  not  in  C++
```

因为函数声明应当更加清晰，所以这是一个改进。但不幸的是，我不能总是通过查看程序代码来知道函数是否有异常抛出 —— 例如，函数的异常抛出发生在存储分配过程中。较为糟糕的是由于调用了在异常处理之前引入的函数而出现非有意的异常抛出。（函数可能与一个新版本的异常抛出相连接）所以采用不明确的描述，如：

```
void   f ( ) ;
```

表示有可能有异常抛出，也可能没有。这种不明确的描述对于避免阻碍程序执行是十分必要的。

17.3.5 捕获所有异常

前面论述过，如果函数没有异常规格说明，任何类型的异常都有可能被函数抛出。为了解决这个问题，应创建一个能捕获任意类型的异常的处理器。这可以通过将省略号加入参数列表(á la C)中来实现这一方案。

```
catch (. . . ) {
    cout <<  "an exception was thrown" <<endl;
}
```

为了避免漏掉异常抛出，可将能捕获任意异常的处理器放在一系列处理器之后。

在参数列表中加入省略号可捕获所有的异常，但使用省略号就不可能有参数，也不可能知道所接受到的异常为何种类型。

17.3.6 异常的重新抛出

有时需要重新抛出刚接收到的异常，尤其是在我们无法得到有关异常的信息而用省略号捕获任意的异常时。这些工作通过加入不带参数的 throw就可完成：

```
catch (. . .) {
    cout <<  "an exception was thrown "<<endl;
    throw;
}
```

如果一个catch句子忽略了一个异常，那么这个异常将进入更高层的上下文环境。由于每个异常抛出的对象是被保留的，所以更高层上下文环境的处理器可从抛出来自这个对象的所有信息。

17.3.7 未被捕获的异常

如果测试块后面的异常处理器没有与某一异常相匹配，这时内层对异常的捕获失败，异常将进入更高层的上下文环境中（高层测试块一般不最先进行异常接收），这个过程一直进行直到在某个层次异常处理器与该异常相匹配，这时这个异常才被认为是被捕获了，进一步的查询也将停止。

假如任意层的处理器都没有捕获到这个异常，那么这个异常就是“未捕获的”或“未处理的”。如果已存在的异常在被捕获之前又有一个新的异常产生将造成异常不能被获取，最常见的这种情况的产生原因是异常对象的构造函数自身会导致新的异常。

1. terminate( )

如果异常未能被捕获，特殊函数terminate( )将自动被调用。如同函数unexception( )终止函数一样，它实际上也是一个指向函数的指针。在C标准库中它的缺省值为指向函数abort( )的指针，abort( )函数可以不用调用正常的终止函数而直接从程序中退出（这意味着静态全局函数的析构函数不用被调用）。

如果一个异常未被捕获，析构函数不会被调用，则异常将不会被清除。含有未捕获的异常将被认为是程序错误。我们可将程序（如果有必要，包括main( )的所有代码）封装在一个测试块中，这个测试块由各异常处理器按序组成，并可以捕获任意异常的缺省处理器(catch(. . .))结束。如果我们不将程序按上述方法封装，将使我们的程序十分臃肿。一个未能被捕获的异常可看成是一个程序错误。

2. set_terminate( )

我们可以使用标准函数set_ terminate( )来安装自己的终止函数terminate( )，set_ terminate( )返回被替代的terminate( )函数的指针，这样就可存贮该指针并在需要时进行恢复。定做的终止函数terminate( )必须不含有输入参数，其返回值为 void。另外所安装的任何终止处理器terminate( )必须不返回或抛出异常，但是作为替换将调用一些程序终止函数。在实际中如果函数terminate( )被调用就意味着问题将无法被恢复。

如同函数unexpected( )一样，函数terminate( )的指针不能为零。

这儿给出一实例用以展示set_ terminate( )的使用。例中set_ terminate( )函数被调用后返回函数terminate( )的原先的指针，存储该指针并为以后的恢复做准备，这样可通过函数terminate( )为判断未捕获的异常在程序中何处发生提供帮助：

```
//: TRMNATOR.CPP -- Use of set_terminate()
// Also shows uncaught exceptions
#include <except.h>
#include <iostream.h>
#include <stdlib.h>

void terminator() {
  cout << "I'll be back!" << endl;
  abort();
}
```

```
void (*old_terminate)()
  = set_terminate(terminator);

class botch {
public:
  class fruit {};
  void f() {
    cout << "botch::f()" << endl;
    throw fruit();
  }
  ~botch() { throw 'c'; }
};

main() {
  try{
    botch b;
    b.f();
  } catch(...) {
    cout << "inside catch(...)" << endl;
  }
}
```

old_terminate的定义初看上去有些令人费解：该语句不仅创建了一个指向函数的指针old_terminate，而且将其初始化为函数set_ terminate( )的返回值。虽然我们可能比较熟悉在函数指针后面加分号的定义方法，但例中所给出是另一种变量并可在定义时进行初始化。

类botch不仅在函数f( )内部会抛出异常，而且在它的析构函数内也会抛出异常。从主程序中可见，这是调用函数terminate( )的一种情况。虽然异常处理器中使用了catch(...)函数，从表面上看它似乎可以捕获所有的异常，避免函数terminate( )的调用，但是当处理一个异常需清除堆栈中的对象时，在这一过程中将调用类botch的析构函数，由此产生了第二个异常，这将迫使函数terminate( )被调用。因此析构函数中含有异常抛出或引起异常抛出都将是一个设计错误。

17.4 清除

异常处理的部分难度就在于异常抛出时从正常程序流转入异常处理器中。如果异常抛出时对象没有被正确地清除，这一操作将不会很有效。 C++的异常处理器可以保证当我们离开一个作用域时，该作用域中所有结构完整的对象的析构函数都将被调用，以清除这些对象。

这里给出一个例子，用以演示当对象的构造函数不完整时其析构函数将不被调用，它也用来展示如果在被创建对象过程中发生异常抛出时将出现什么结果，如果 unexpected( )函数再次抛出意外的异常时将出现什么结果：

```
//: CLEANUP.CPP -- Exceptions clean up objects
#include <fstream.h>
#include <except.h>
#include <string.h>

ofstream out("cleanup.out");
```

```
class noisy {
  static int i;
  int objnum;
  enum { sz = 40 };
  char name[sz];
public:
  noisy(const char* nm="array elem") throw(int){
    objnum = i++;
    memset(name, 0, sz);
    strncpy(name, nm, sz - 1);
    out << "constructing noisy " << objnum
      << " name [" << name << "]" << endl;
    if(objnum == 5) throw int(5);
    // Not in exception specification:
    if(*nm == 'z') throw char('z');
  }
  ~noisy() {
    out << "destructing noisy " << objnum
      << " name [" << name << "]" << endl;
  }
  void* operator new[](size_t sz) {
    out << "noisy::new[]" << endl;
    return ::new char[sz];
  }
  void operator delete[](void* p) {
    out << "noisy::delete[]" << endl;
    ::delete []p;
  }
};

int noisy::i = 0;

void unexpected_rethrow() {
  out << "inside unexpected_rethrow()" << endl;
  throw; // Rethrow same exception
}
main() {
  set_unexpected(unexpected_rethrow);
  try {
    noisy n1("before array");
    // Throws exception:
    noisy* array = new noisy[7];
    noisy n2("after array");
  } catch(int i) {
    out << "caught " << i << endl;
  }
```

```
  out << "testing unexpected:" << endl;
  try {
    noisy n3("before unexpected");
    noisy n4("z");
    noisy n5("after unexpected");
  } catch(char c) {
    out << "caught " << c << endl;
  }
}.
```

类noisy可跟踪对象的创建，所以可通过它跟踪程序的运行。类 noisy中含有静态整数变量i用以记录创建对象的个数，整数变量 objnum用以记录特殊对象的个数，字符缓冲器 name用以保存字符标识符。该缓冲器首先设置为零，然后把构造函数的参数拷贝给它。（注意这里用缺省的字符串参数表明所创建的为数组元素，所以该构造函数实际上充当了缺省构造函数。）因为C标准库中函数strncpy( )在它的第三个参数指定的字符数出现或零终结符出现时，将终止字符的复制，所以被复制字符的数肯定小于缓冲器的大小，并且最后一个字符始终为零，因此打印语句将决不会超出缓冲器。

构造函数在两种情况下会发生异常抛出。第一种情况是当第五个对象被创建时（这只是为了显示在对象数组创建中发生异常，而不是真正的异常条件），这种异常将抛出一个整数，并且函数在异常规格说明中已引入了整数类型。第二种情况当然也是特意设计的，当参数字符串的第一个字符为“ z ”时将抛出一字符型异常。由于异常规格说明中不含有字符型，所以这类异常将调用unexpected( )函数。

函数new 和 delete可对类进行重载，其功能可见其函数调用。

函数unexpected_rethrow( ) 打印一条信息，并且再次抛出同一个异常。在主程序 main( )的第一行中，它充当unexpected( )函数被安装。在测试块中将创建一些noisy对象，但是在对象数组的创建中有异常抛出，所以对象n2将不会被创建。这些在程序输出结果中可以见到：

```
constructing noisy 0 name [before array]
noisy::new[]
constructing noisy 1 name [array elem]
constructing noisy 2 name [array elem]
constructing noisy 3 name [array elem]
constructing noisy 4 name [array elem]
constructing noisy 5 name [array elem]
destructing noisy 4 name [array elem]
destructing noisy 3 name [array elem]
destructing noisy 2 name [array elem]
destructing noisy 1 name [array elem]
noisy::delete[]
destructing noisy 0 name [before array]
caught 5
testing unexpected:
constructing noisy 6 name [before unexpected]
constructing noisy 7 name [z]
inside unexpected_rethrow()
destructing noisy 6 name [before unexpected]
```

```
caught z
```

程序成功地创建了四个对象数组单元，但在构造第五个对象时发生异常抛出。由于第五个对象的构造函数未完成，因此异常在清除对象时只有1~4的析构函数被调用。

全局函数new的一次调用所产生的对象数组的存储分配是分离的。注意，即使程序中没有明确地调用函数delete，但异常处理系统仍不可避免地调用delete函数来释放存储单元。只有在使用规范的new函数形式时才会出现上述情况。如果使用第12章介绍的语法，异常处理机构将不会调用delete函数来清除对象，因为它只适用于清除不是堆结构的存储区。

最终对象n1将被清除，而对象n2由于没被创建所以也不存在被清除的问题。

在测试函数unexpected_rethrow( )的程序段中，对象n3已被创建，对象n4的构造函数已开始创建对象。但是在它创建完成之前已有异常抛出。该异常为字符型，不存在于函数的异常规格说明中，所以函数unexpection( )将被调用（在此例中为函数unexpected_rethrow( )）。由于函数unexpected_rethrow( )可抛出所有类型的异常，所以该函数将再次抛出与已知类型完全相同的异常。当对象n4的构造函数被调用抛出异常后，异常处理器将进行查找并捕获该异常（在成功创建的对象n3被清除之后）。这样函数unexpected_rethrow( )的作用就是接收任意的未加说明的异常，并作为已知异常再次抛出；使用这种方法该函数可为我们提供一滤波器，用以跟踪意外异常的出现并获取该异常的类型。

17.5 构造函数

当编写的程序出现异常时，我们总会问：“当异常出现时，这能被合理地清除掉吗？”这是十分重要的。对于大多数情况，程序是相当安全的；但是如果构造函数中出现异常，这将产生问题：如果异常抛出发生在构造函数创建对象时，对象的析构函数将无法调用其相应的对象。这意味着在编写构造函数的程序时必须十分谨慎。

构造函数进行存储资源分配时存在普遍的困难。如果构造函数在运行时有异常抛出，析构函数将无法收回这些存储资源。这些问题大多数发生在未加保护的指针上。例如：

```
//: NUDEP.CPP -- Naked pointers
#include <fstream.h>
#include <stdlib.h>
ofstream out("nudep.out");

class bonk {
public:
  bonk() { out << "bonk()" << endl; }
  ~bonk() { out << "~bonk()" << endl; }
};

class og {
public:
  void* operator new(size_t sz) {
    out << "allocating an og" << endl;
    throw int(47);
    return 0;
  }
  void operator delete(void* p) {
```

```
    out << "deallocating an og" << endl;
    ::delete p;
  }
};

class useResources {
  bonk* bp;
  og* op;
public:
  useResources(int count = 1) {
    out << "useResources()" << endl;
    bp = new bonk[count];
    op = new og;
  }
  ~useResources() {
    out << "~useResources()" << endl;
    delete []bp; // Array delete
    delete op;
  }
};

main() {
  try {
    useResources ur(3);
  } catch(int) {
    out << "inside handler" << endl;
  }
};
```

输出是：

```
useResources()
bonk()
bonk()
bonk()
allocating an og
inside handler
```

当进入类useResources的构造函数后，并且类bonk的构造函数已成功地完成了对象数组的创建，而这时，在og::operator new中抛出一个异常（例如存储耗尽所产生的异常）。这样我们就意外地在异常处理器中结束程序，而 useResources所对应的析构函数未得到调用。这是正常的，因为类useResources的构造函数的全部构造工作没能全部完成，这就意味着基于堆存储的类bonk的对象也不能被析构。

对象化

为了防止上文提到的情况，应避免对象通过本身的构造函数和析构函数将“不完备的”资源分配到对象中。利用这种方法，每个分配就变成了原子的，像一个对象，并且如果失败，那

么已分配资源的对象也被正确地清除。采用模板是修改上例的一个好方法：

```
//: WRAPPED.CPP -- Safe, atomic pointers
#include <fstream.h>
#include <stdlib.h>
ofstream out("wrapped.out");

// Simplified. Yours may have other arguments.
template<class T, int sz = 1> class pwrap {
  T* ptr;
public:
  class rangeError {}; // Exception class
  pwrap() {
    ptr = new T[sz];
    out << "pwrap constructor" << endl;
  }
  ~pwrap() {
    delete []ptr;
    out << "pwrap destructor" << endl;
  }
  T& operator[](int i) throw(rangeError) {
    if(i >= 0 && i < sz) return ptr[i];
    throw rangeError();
  }
};

class bonk {
public:
  bonk() { out << "bonk()" << endl; }
  ~bonk() { out << "~bonk()" << endl; }
  void g() {}
};

class og {
public:
  void* operator new[](size_t sz) {
    out << "allocating an og" << endl;
    throw int(47);
    return 0;
  }
  void operator delete[](void* p) {
    out << "deallocating an og" << endl;
    ::delete p;
  }
};

class useResources {
```

```
  pwrap<bonk, 3> Bonk;
  pwrap<og> Og;
public:
  useResources() : Bonk(), Og() {
    out << "useResources()" << endl;
  }
  ~useResources() {
    out << "~useResources()" << endl;
  }
  void f() { Bonk[1].g(); }
};

main() {
  try {
    useResources ur;
  } catch(int) {
    out << "inside handler" << endl;
  } catch(...) {
    out << "inside catch(...)" << endl;
  }
}
```

不同点是使用模板封装指针并将它送入对象。这些对象的构造函数的调用先于useResources构造函数的调用，如果这些构造函数的创建操作完成之后发生了异常抛出，与它们相对应的析构函数被调用。

模板pwrap演示了比前面所见更为经典的异常使用：如果 operator[]的参数出界，那么就创建一个嵌入类rangeError用于operator[]中。因为operator[]返回一个引用而不是返回0。（没有0引用。）这是一个真实的异常情况：不知道在当前上下文中该做什么，也不能返回一个不可能的值。在此例中，rangeError很简单而且设想所有必须的信息都在类名中，但我们也可加入含有索引值的成员，如果这样做有用的话。

现在输出是：

```
bonk()
bonk()
bonk()
pwrap constructor
allocating an og
~bonk()
~bonk()
~bonk()
pwrap destructor
inside handler
```

对og的空间存储分配又抛出一个异常，但这次bonk对象数组正确地被清除，所以没有存储损耗。

17.6 异常匹配

当一个异常抛出时，异常处理系统会根据所写的异常处理器顺序找到“最近”的异常处理器，而不会搜寻更多的异常处理器。

异常匹配并不要求在异常和处理器之间匹配得十分完美。一个对象或一个派生类对象的引用将与基类处理器匹配（然而假若处理器针对的是对象而非引用，异常对象在传递给处理器时会被“切片”，这样不会受到破坏但会丢失所有的派生类型信息）。假若抛出一个指针，标准指针转化处理会被用于匹配异常，但不会有自动的类型转化将某个异常类型在匹配过程中转化为另一个。下面是一个例子：

```
//: AUTOEXCP.CPP -- No matching conversions
#include <iostream.h>
class except1 {};
class except2 {
public:
  except2(except1&) {}
};

void f() { throw except1(); }

main() {
  try { f();
  } catch (except2) {
    cout << "inside catch(except2)" << endl;
  } catch (except1) {
    cout << "inside catch(except1)" << endl;
  }
}
```

尽管我们可能认为第一个处理器会使用构造函数转化，将一个 except1对象转化成except2对象，但是系统在异常处理期间将不会执行这样的转换，我们将在 except1处终止。

下面的例子展示基类处理器怎样捕获派生类的异常：

```
//: BASEXCPT.CPP -- Exception hierarchies
#include <iostream.h>

class X {
public:
  class trouble {};
  class small : public trouble {};
  class big : public trouble {};
  void f() { throw big(); }
};

main() {
  X x;
  try {
```

```
    x.f();
  } catch(X::trouble) {
    cout << "caught trouble" << endl;
  // Hidden by previous handler:
  } catch(X::small) {
    cout << "caught small trouble" << endl;
  } catch(X::big) {
    cout << "caught big trouble" << endl;
  }
}
```

这里的异常处理机制，对于第一个处理器总是匹配一个 trouble对象或从trouble派生的什么事物，由于第一个处理器捕获涉及第二和第三处理器的所有异常，所以第二和第三处理器永远不被调用。光捕获派生类异常把基类的一般异常放在末端捕获更有意义（或者在随后的下一个开发周期中引入的派生类）。

另外，假若small和big的对象比trouble的大（这常常是真实的，因为通常为派生类添加成员），那么这些对象会被“切片”以适应处理器。当然，在本例中由于派生类没有附加成员，而且在处理器中也没有参数标识，所以这一点并不重要。通常在处理器中，应该使用引用参数而非对象以避免裁剪掉信息。

17.7 标准异常

用于C++类标准库的一批异常可以用于我们自己的程序中。从标准异常类开始会比我们尽量自己定义来得快和容易。假若标准异常类不能满足需要，我们可以继承它并添加自己的特定内容。下面的表描述了标准异常：

exception	是所有标准C++库异常的基类。我们可以调用 what()以获得其特性的显示说明
logic_erro	是由exception派生的。它报告程序的逻辑错误，这些错误在程序执行前可以被检测到
runtime_error	是由exception派生的。它报告程序运行时错误，这些错误仅在程序运行时可以被检测到

I/O流异常类ios::failure也由exception派生，但它没有进一步的子类：

下面两张表中的类都可以按说明使用，也可以作为基类去派生我们自己的更为特殊的异常类型。

由logic_error派生的异常	
domain_error	报告违反了前置条件
invalid_argument	指出函数的一个无效参数
length_error	指出有一个产生超过NPOS长度的对象的企图（NPOS：类型size_t 的最大可表现值）
out_of_range	报告参数越界
bad_cast	在运行时类型识别中有一个无效的 dynamic_cast表达式（见第 18章）
bad_typeid	报告在表达式typeid(*p)中有一个空指针 P（运行时类型识别的特性见第 18章）

由runtime_error派生的异常	
range_error	报告违反了后置条件
overflow_error	报告一个算术溢出
bad_alloc	报告一个存储分配错误

17.8 含有异常的程序设计

对大多数程序员尤其是C程序员，在他们的已有的程序设计语言中不能使用异常，需进一步矫正。下面是一些含有异常的程序设计原则。

17.8.1 何时避免异常

异常并不能回答所发生的所有问题。实际上若对异常进行钻牛角尖式的推敲，将会遇到许多麻烦。下面的段落指出异常不能被保证的情况。

1. 异步事件

标准C的signal()系统以及其他类似的系统操纵着异步事件：该事件发生在程序控制的范围以外，它的发生是程序所不能预计的。由于异常和它的处理器都在相同的调用栈上，所以异常不能用来处理异步事件。也就是异常限制在某范围内，而异步事件必须有完全独立的代码来处理，这些代码不是普通程序流的一部分（典型的如中断服务和事件循环例程）。

这并不是说异步事件不能和异常发生关系。但是，中断服务处理器都尽可能快地工作，然后返回。在一些定义明确的程序点上，一个异常可以以基于中断的方式抛出。

2. 普通错误情况

假若有足够的信息去处理一个错误，这个错误就不是一个异常。我们应该关心当前的上下文环境，而不应该把异常抛向更大的上下文环境中。同样，在 C++中也不应当为机器层的事件抛出异常，如“ 除零溢出 ”。可以认为这些“ 异常 ”可由其他的机制去处理，如操作系统或硬件。这样，C++异常可以相当有效，并且它们的使用和程序级的异常条件相互隔离。

3. 流控制

一个异常看上去有点象一个交替返回机制，也有点象一个 switch语句段，我们可能被它们吸引，改变了想使用它们的初衷，这是一个很糟糕的想法，一部分原因是因为异常处理系统比普通的程序运行缺乏效率。异常是一个罕有的事件，所以普通程序不应为其支付时间，来自非错误条件的其他什么地方的异常也会给使用我们的类或函数的用户带来相当的混乱。

4. 不强迫使用异常

一些程序相当简单，如一些实用程序，可能仅仅需要获取输入和执行一些加工。如果在这类程序中试图分配存储然而失败了，或打开一个文件然而失败了等等，这样可以在这类程序中使用assert()以示出错信息，使用abort()终止程序，允许系统清除混乱。但是如果我们自己努力去捕获所有异常，修复系统资源，则是不明智的。从根本上说，假若我们不必使用异常，我们就不要用。

5. 新异常，老代码

另一种情形出现在对没有使用异常的已存在的程序进行修改的时候。我们可能引入一个使用异常的库而且想知道是否有必要在程序中修改所有的代码。假定已经安放了一个可接受的出错处理配置，这里所要做的最明智的事情是围绕着使用新类 try块的最大程序块，追加一个catch(…)和基本出错信息。我们可以追加有必要的更多特定的处理器，并使修改更为细致。但是，在这种情况下，被迫增加的代码必须是最小限度的。

我们也可以把我们的异常生成代码隔离在try块中，并且编写一个把当前异常转换成已存在的出错处理方案的处理器。

创建一个为其他人使用的库，而且无从知晓用户在遭遇决定性错误的情况下如何反应，这时考虑异常才真正重要。

17.8.2 异常的典型使用

使用异常便于：

1) 使问题固定下来和重新调用这个（导致异常的）函数。

2) 把事情修补好而继续运行，不去重试函数。

3) 计算一些选择结果用于代替函数假定产生的结果。

4) 在当前上下文环境尽其所能并且再把同样的异常弹向更高的上下文中。

5) 在当前上下文环境尽其所能并且把一个不同的异常弹向更高的上下文中。

6) 终止程序。

7) 包装使用普通错误方案的函数（尤其是C的库函数），以便产生异常替代。

8) 简化，假若我们的异常方案建造得过于复杂，使用时会令人懊恼。

9) 使我们的库和程序更安全。这是短期投资（为了调试）和长期投资（为了应用的健壮性）问题。

1. 随时使用异常规格说明

异常的规格说明像一个函数原型：它告诉用户书写异常处理代码以及处理什么异常。它告诉编译器异常可能出现在这个函数中。

当然，我们不能总是通过检查代码而预见什么异常会发生在特定的函数中。有时这个特定函数所调用的函数产生了一个出乎意料的异常，有时一个不会抛出异常的老函数被一个会抛出异常的新函数替换了，这样我们将产生对 unexpect()的调用。无论何时，只要使用异常规格说明或者调用含有异常的函数，都应该创建自己的 unexpected()函数，该函数记录信息而且重新抛出同样的异常。

2. 起始于标准异常

在创建我们自己的异常前应检查标准 C++异常库。假若标准异常正合所需，则这样会使我们的用户更易于理解和处理。

假若所需要的异常类型不是标准库的一部分，则尽量从某个已存在标准 expection中派生形成。假若在expection的类接口中总是存在what()函数的期望定义，这会使用户受益匪浅。

3. 套装我们自己的异常

如果为我们的特定类创建异常，在我们的类中套装异常类是一个很好的主意，这为读者提供了一个清晰的消息 —— 这些异常仅为我们的类所使用。另外，它可防止命名域混乱。

4. 使用异常层次

异常层次为不同类型的重要错误的分类提供了一个有价值的方法，这些错误可能会与我们的类或库冲突。该层次可为用户提供有帮助的信息，帮助他们组织自己的代码，让他们可以选择是忽略所有异常的特定类型还是正确地捕获基类类型。而且在以后，任何异常可通过对相同基类的继承而追加，而不会被迫改写所有的已生成代码 —— 基类处理器将捕获新的异常。

当然，标准C++异常是一个异常层次的优秀例子，通过使用可进一步增强和丰富它。

5. 多重继承

我们会记得，在第15章中，多重继承最必要做的地方就是需要把一个指向对象的指针向上映射到两个不同的基类，也就是需要两个基类的多态行为的地方。这样，异常层次对于多重继承是有用的，因为多重继承异常类的任一根的基类处理器都可处理异常。

6. 用“引用”而非“值”去捕获

如果抛出一个派生类对象而且该对象被基类的对象处理器通过值捕获到，对象会被“切片”，

这就是说，随着向基类对象的传递，派生类元素会依次被割下，直到传递完成。这样的偶然性并不是所要的，因为对象的行为像基类而不象它本来就是的派生类对象（实际就是“切片”以前）。下面是一个例子：

```
//: CATCHREF.CPP -- Why catch by reference?
#include <iostream.h>

class base {
public:
  virtual void what() {
    cout << "base" << endl;
  }
};

class derived : public base {
public:
  void what() {
    cout << "derived" << endl;
  }
};

void f() { throw derived(); }

main() {
  try {
    f();
  } catch(base b) {
    b.what();
  }
  try {
    f();
  } catch(base& b) {
    b.what();
  }
}
```

输出为

```
base
derived
```

当对象通过值被捕获时,因为它被转化成一个base对象(由构造函数完成)，而且在所有的情况下表现出base对象的行为；然而当对象通过引用被捕获时，仅仅地址被传递而对象不会被切片，所以它的行为反映了它处于派生中的真实情况。

虽然也可以抛出和捕获指针，但这样做会引入更多的耦合——抛出器和捕获器必须为怎样分配和清理异常对象而达成一致。这是一个问题，因为异常本身可能会由于堆的耗尽而产生。如果抛出异常对象，异常处理系统会关注所有的存储。

7. 在构造函数中抛出异常

由于构造函数没有返回值，因此在先前我们可以有两个选择以报告在构造期间的错误：

1) 设置一个非局部标志并且希望用户检查它。

2) 返回一个不完全被创建的对象并且希望用户检查它。

这是一个严重的问题，因为 C程序员必须依赖一个隐含的保证：对象总是成功地被创建，这在类型如此粗糙的C中是不合理的。但是在C++程序中，构造失败后继续执行是注定的灾难，于是构造函数成为抛出异常最重要的地方之一。现在有一个安全有效的方法去处理构造函数错误。然而我们还必须把注意力集中在对象内部的指针上和构造函数异常抛出时的清除方法上。

8. 不要在析构函数中导致异常

由于析构函数会在抛出其他异常时被调用，所以永远不要打算在析构函数中抛出一个异常，或者通过执行在析构函数中的相同动作导致其他异常的抛出。如果这些发生了，这意味着在已存在的异常到达引起捕获之前抛出了一个新的异常，这会导致对 terminate()的调用。

这里的意思是：假若调用一个析构函数中的任何函数都有可能会抛出异常，这些调用应该写在析构函数中的一个try块中，而且析构函数必须自己处理所有自身的异常。这里的异常都不应逃离析构函数。

9. 避免无保护的指针

请看第17.5.1节中的WRAPPED.CPP程序，假若资源分配给无保护的指针，那么意味着在构造函数中存在一个缺点。由于该指针不拥有析构函数，所以当在构造函数中抛出异常时那些资源将不能被释放。

17.9 开销

为了使用新特性必然有所开销。当异常被抛出时有相当的运行时间方面的开销，这就是从来不想把异常用于普通流控制的一部分的原因，而不管它多么令人心动。异常的发生应当是很少的，所以开销聚集在异常上而不是在普通的执行代码上。设计异常处理的重要目标之一是：在异常处理实现中，当异常不发生时应不影响运行速度。这就是说，只要不抛出异常，代码的运行速度如同没有加载异常处理时一样。无论与否，异常处理都依赖于使用的特定编译器。

异常处理也会引出额外信息，这些信息被编译器置于栈上。

除了能作为特定的“ 异常范围 ”（它可能恰恰是全局范围）的对象传进送出外，异常对象可以像其他对象一样被正确地在周围传递。当异常处理器工作完成时，异常对象也被相应地销毁。

17.10 小结

错误恢复是和我们编写的每个程序相关的基本原则，在 C++中尤其重要，创建程序组件为其他人重用是开发的目标之一。为了创建一个稳固系统，必须使每个组件具有健壮性。

C++中异常处理的目标是简化大型可靠程序的创建，使用尽可能少的代码，使应用中没有不受控制的错误而使我们更加自信。这几乎不损害性能，并且对其他代码的影响很小。

基本异常不特别难学，我们应该在程序中尽量地使用它们。异常是能给我们提供即时而显著的好处的特性之一。

17.11 练习

1) 创建一个含有可抛出异常的成员函数的类。在该类中，创建一个被嵌套的类用作一个异

常对象，它带有一个char*参数，该参数表示一个描述型字符串。创建一个可抛出该异常的成员函数。（标明函数的异常规格说明）书写一个try块使它能调用该函数并且捕获异常，以打印描述型字符串的方式处理该异常。

2) 重写第12章中的stash类以便为operator[]抛出out-of-range异常。

3) 写一个一般的main()，它可取走所有的异常并且报告错误。

4) 创建一个拥有自身运算符new的类。该运算符分配10个对象，在对第11个对象分配时假定“存储耗尽”并抛出一个异常。增加一个静态函数用于回收存储。现在，创建一个伴有try块和能够调用存储恢复例程的catch子句的主程序，将这些都放入一个while循环中，演示异常恢复和连续执行的情形。

5) 创建一个可抛出异常的析构函数，编写代码以向自己证明这是一个糟糕的想法。该代码可展示如果处理器对一个已存在异常施加影响之前一个新异常又抛出了，那么terminate()会被调用。

6) 向我们自己证明所有的异常对象（被抛出的）都能被正确地销毁。

7) 向我们自己证明假若我们在堆上创建一个异常对象并且抛出一个指向该对象的指针，则它不会被清理掉。

8) （高级）。使用一个带有构造函数和拷贝构造函数的类来追踪异常的创建和传递，这些构造函数和拷贝构造函数显示它们自身而且尽可能地提供关于对象是怎样创建的信息（就拷贝构造函数而言，说明创建什么对象）。创立一个有趣的状态，抛出我们的新类型的对象并分析结果。

第18章　运行时类型识别

运行时类型识别（Run-time type identification, RTTI）是在我们只有一个指向基类的指针或引用时确定一个对象的准确类型。

这可以被看作C++的第二大特征，在我们茫然不知所措时，这确实是一个很有用的工具。一般情况下，我们并不需要知道一个类的确切类型，虚函数机制可以实现那种类型的正确行为。但是有些时候，我们有指向某个对象的基类指针，确定该对象的准确类型是很有用的。这些信息让我们更高效地完成一个特定情况下的操作，防止基类的接口变得很笨拙。大多数的类库用一些虚函数来提供运行时的类型信息。当异常处理功能加入到 C++时，它要求知道有关这个对象的准确类型信息。下一步是在语言中访问这些信息，这很容易实现。

这一章解释RTTI是干什么的以及怎样使用它。另外，这一章还解释了 C++新的映射语法是什么，它和RTTI很相似。

18.1　例子——shape

这里有一个使用多态性的类层次的例子。基类为 shape，三个派生类分别为circle、square和triangle；

下面是一个典型的类继承关系图，基类在上，派生类向下生长。面向对象程序设计的一般目标就是用代码体管理指向基类的指针，所以如果想增加一个新类来扩充程序（比如从 shape中派生出rhomboid），代码体部分并不受影响。在本例中， shape接口部分的虚函数是draw()，其目的就是让用户程序员通过一个shape指针来调用draw()，draw()在所有的派生类中都被重新定义。由于它是一个虚函数，所以即使是用一个 shape()型的指针来调用它，它仍然会被正确调用。

因此，创建一个特定的对象 (circle、square或triangle)，取它的地址并把它映射到 shape*（忘掉对象的实际类型），然后在程序的其他地方用这个匿名指针。继承关系图如上所示，所以这种从多个派生类到基类的映射叫做向上映射。

18.2　什么是RTTI

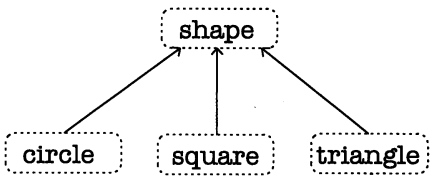


图　18-1

假如在编程中遇到了特殊的问题，而只要我们知道了一个一般指针的准确类型它就会迎刃而解，我们该怎么办？比如，假设允许我们的用户将任一形状变成紫色来表示加亮。用这种方法，他们可以发现屏幕上的所有三角形都被加亮。我们可能自然地想到用虚函数，像 TurnColorIfYouAreA()，它允许一些种类颜色的枚举型参数和 shape::circle、shape::square或shape::triangle参数。

为了解决这种问题，多数类库设计者把虚函数放在基类中，使运行时返回特定对象的类型信息。我们可能见过一些名字为 isA()和type Of() 之类的成员函数，这些就是开发商定义的RTTI函数。使用这些函数，当处理一个对象列表时就可以说：“ 如果这个对象是triangle类的，

就把它变成紫色。”

当C++中引进异常处理时，它的实现要求把一些运行时间类型信息放在虚函数表中。这意味着只要对语言作一点小小的扩充，程序员就能获得有关一个对象的运行时间类型信息。所有的开发商都在自己的类库中加入了RTTI，所以它已包含在C++语言中。

RTTI与异常一样，依赖驻留在虚函数表中的类型信息。如果试图在一个没有虚函数的类上用RTTI，就得不到预期的结果。

RTTI的两种使用方法

使用RTTI有两种不同的方法。第一种就像sizeof()，因为它看上就像一个函数。但实际上它是由编译器实现的。typeid()带有一个参数，它可以是一个对象引用或指针，返回全局typeinfo类的常量对象的一个引用。可以用运算符“==”和“!=”来互相比较这些对象。也可以用name()来获得类型的名称。注意，如果给typeid()传递一个shape*型参数，它会认为类型为shape*，所以如果想知道一个指针所指对象的精确类型，我们必须逆向引用这个指针。比如，s是个shape*，

```
cout << typeid(*s).name()<<endl;
```

将显示出s所指向的对象类型。

也可以用before(typeinfo&)查询一个typeinfo对象是否在另一个typeinfo对象的前面（以定义实现的排列顺序），它将返回true或false。如果写：

```
if(typeid(me).before(typeid(you))) //...
```

那么表示我们正在查询me在排列顺序中是否在you之前。

RTTI的第二个用法叫“安全类型向下映射”。之所以用“向下映射”这个词也是由于类继承的排列顺序。如果映射一个circle*到shape*叫向上映射的话，那么将一个shape*映射成一个circle*就叫向下映射了。当然一个circle*也是一个shape*，编译器允许任意的向上映射，但一个shape*不一定就是circle*，所以编译器在没有明确的类型映射时并不允许我们完成一个向下映射任务。当然可以用原有的C风格的类型映射或C++的静态映射（static_cast,将在本章末介绍）来强制执行，这等于在说：“我希望它实际上是一个circle*，而且我打算要求它是。”由于并没有明确地知道它实际上是circle，因此这样做是很危险的。在开发商制定的RTTI中一般的方法是：创建一个函数来试着将shape*指派为一个circle*(在本例中)，检查执行过程中的数据类型。如果这个函数返回一个地址，则成功；如果返回null，说明我们并没有一个circle*对象。

C++的RTTI的“安全类型向下映射”就是按照这种“试探映射”函数的格式，但它（非常合理地）用模板语法来产生这个特殊的动态映射函数（dynamic_cast），所以本例变成：

```
shape* sp = new circle;
circle* cp = dynamic_cast<circle*>(sp);
if(cp) cout << "cast successful";
```

动态映射的模板参数是我们想要该函数创建的数据类型，也就是这个函数的返回值。函数参数是我们试图映射的源数据类型。

通常只要对一种类型作这种工作（比如将三角型变成紫色），但如果想算出各种shape的数目，可以用面下例子的框架：

```
circle* cp = dynamic_cast<circle*>(sh);
square* sp = dynamic_cast<square*>(sh);
triangle* tp = dynamic_cast<triangle*>(sh);
```

当然这是人为的——我们可能已经在各个类型中放了一个静态数据成员并在构造函数中对

它自增。如果可以控制类的源代码并可以修改它，当然可以这样做。下面这个例子用来计算shape的个数，它用了静态数据成员和动态映射两种方法：

```
//: RTSHAPES.CPP -- Counting shapes
#include <iostream.h>
#include <time.h>
#include <typeinfo.h>
#include "..\14\tstash.h"

class shape {
protected:
  static int count;
public:
  shape() { count++; }
  virtual ~shape() = 0 { count--; }
  virtual void draw() const = 0;
  static int quantity() { return count; }
};

int shape::count = 0;

class rectangle : public shape {
  void operator=(rectangle&); // Disallow
protected:
  static int count;
public:
  rectangle() { count++; }
  rectangle(const rectangle&) { count++;}
  ~rectangle() { count--; }
  void draw() const {
    cout << "rectangle::draw()" << endl;
  }
  static int quantity() { return count; }
};

int rectangle::count = 0;

class ellipse : public shape {
  void operator=(ellipse&); // Disallow
protected:
  static int count;
public:
  ellipse() { count++; }
  ellipse(const ellipse&) { count++; }
  ~ellipse() { count--; }
  void draw() const {
    cout << "ellipse::draw()" << endl;
```

```
  }
  static int quantity() { return count; }
};

int ellipse::count = 0;

class circle : public ellipse {
  void operator=(circle&); // Disallow
protected:
  static int count;
public:
  circle() { count++; }
  circle(const circle&) { count++; }
  ~circle() { count--; }
  void draw() const {
    cout << "circle::draw()" << endl;
  }
  static int quantity() { return count; }
};

int circle::count = 0;

main() {
  tstash<shape> shapes;
  time_t t;
  // Seed random number generator:
  srand((unsigned)time(&t));
  const mod = 12;
  for(int i = 0; i < rand() % mod; i++)
    shapes.add(new rectangle);
  for(int j = 0; j < rand() % mod; j++)
    shapes.add(new ellipse);
  for(int k = 0; k < rand() % mod; k++)
    shapes.add(new circle);
  int Ncircles = 0;
  int Nellipses = 0;
  int Nrects = 0;
  int Nshapes = 0;
  for(int u = 0; u < shapes.count(); u++) {
    shapes[u]->draw();
    if(dynamic_cast<circle*>(shapes[u]))
      Ncircles++;
    if(dynamic_cast<ellipse*>(shapes[u]))
      Nellipses++;
    if(dynamic_cast<rectangle*>(shapes[u]))
      Nrects++;
```

```
    if(dynamic_cast<shape*>(shapes[u]))
      Nshapes++;
  }
  cout << endl << endl
       << "circles = " << Ncircles << endl
       << "ellipses = " << Nellipses << endl
       << "rectangles = " << Nrects << endl
       << "shapes = " << Nshapes << endl
       << endl
       << "circle::quantity() = "
       << circle::quantity() << endl
       << "ellipse::quantity() = "
       << ellipse::quantity() << endl
       << "rectangle::quantity() = "
       << rectangle::quantity() << endl
       << "shape::quantity() = "
       << shape::quantity() << endl;
}
```

对于这个例子，两种方法都是可行的，但静态数据成员方法只能用于我们拥有源代码并已安装了静态数据成员和成员函数时（或者开发商已为我们提供了这些），另外RTTI可能在不同的类中用法不同。

18.3 语法细节

本节详细介绍RTTI的两种形式是如何运行的以及两者之间的不同。

18.3.1 对于内部类型的typeid()

为了保持一致性，typeid()也可以运用于内部类型，所以下面的表达式结果为true：

```
typeid(47) == typeid(int)
typeid(0) == typeid(int)
int i;
typeid(i) == typeid(int)
typeid(&i) == typeid(int*)
```

18.3.2 产生合适的类型名字

typeid()必须在所有的状况下都可以运行，比方说，下面的类中包含了一个嵌套类：

```
//: RNEST.CPP -- Nesting and RTTI
#include <iostream.h>
#include <typeinfo.h>

class one {
  class nested {};
  nested* n;
public:
```

```
    one() { n = new nested; }
    ~one() { delete n; }
    nested* N() { return n; }
};

main() {
    one O;
    cout << typeid(*O.N()).name() << endl;
}
```

typeinfo::name()成员函数还是提供了适当的类名，其结果为one::nested。

18.3.3 非多态类型

虽然typeid( )可以运用于非多态类型（基类中没有虚函数），但我们用这种方法获得的信息是值得怀疑的。假设类层次如下：

```
class X {
  int i;
public:
  // ...
};
class Y : public X {
  int j;
public:
  // ...
};
```

如果创建一个派生类对象并向上映射它：

```
X* xp = new Y;
```

typeid()运算符将返回一个结果，但可能不是我们想要的。因为没有非多态机制，所以可以使用静态类型信息：

```
typeid(*xp) == typeid(X)
typeid(*xp) != typeid(Y)
```

一般希望RTTI用于多态类。

18.3.4 映射到中间级

动态映射不仅可用来确定准确的类型，也可用于多层次继承关系中的中间类型。如下例：

```
class d1 {
public:
  virtual void foo() {}
};
class d2 {
public:
  virtual void bar() {}
};
class mi : public d1, public d2 { };
```

```
class mi2 : public mi {};

d2* D2 = new mi2;
mi2* MI2 = dynamic_cast<mi2*>(D2);
mi* MI = dynamic_cast<mi*>(D2);
```

由于多重继承，问题变得更复杂了。如果我们创建了一个 mi2并将向上映射到根类（在这种情况下，从两种可能的根类中选出一种），然后成功地动态映射回派生类mi或mi2。

我们甚至可以从一个根类映射到另一个：

```
d1* D1 = dynamic_cast<d1*>(D2);
```

这可以成功地映射，因为D2实际上指向一个mi2对象，它包含了类型d1的一个子对象。

映射到中间级在dynamic_cast与typeid()之间产生了一个有趣的差异。typeid()总是产生一个typeinfo 对象的引用来描述一个对象的准确类型，因此它不会给出中间层次的信息。在下面的表达式中（它的值是true），typeid()并不像dynamic_cast那样把D2看作一个指向派生类的指针：

```
typeid(D2) != typeid(mi2*)
```

D2的类型就是指针的准确类型：

```
typeid(D2) == typeid(d2*)
```

18.3.5 void指针

运行时类型的识别对一个void型指针不起作用：

```
//: VOIDRTTI.CPP -- RTTI & void pointers
#include <iostream.h>
#include <typeinfo.h>

class stimpy {
public:
  virtual void happy() {}
  virtual void joy() {}
};

main() {
  void* v = new stimpy;
  // Error:
//!  stimpy* s = dynamic_cast<stimpy*>(v);
  // Error:
//!  cout << typeid(*v).name() << endl;
}
```

void*确实意味着“根本没有类型信息”。

18.3.6 用模板来使用RTTI

模板产生许多不同类的名字，而有时希望指出有关下面使用的对象是哪个类的。RTTI提供

了一个实现此功能的方便方法。下面的例子修改了第13章的代码，它没有用预处理宏显示了构造函数和析构函数调用的顺序。

```
//: INHORDER.CPP -- Order of constructor calls
#include <iostream.h>
#include <typeinfo.h>

template<int id> class announce {
public:
  announce() {
    cout << typeid(*this).name()
         << " constructor " << endl;
  }
  ~announce() {
    cout << typeid(*this).name()
         << " destructor " << endl;
  }
};

class X : public announce<0> {
  announce<1> m1;
  announce<2> m2;
public:
  X() { cout << "X::X()" << endl; }
  ~X() { cout << "X::~X()" << endl; }
};

main() { X x; }
```

TYPEINFO.H头文件必须被包含，以便于调用typeid()返回的typeinfo对象的任何成员函数。这个模板用了一个整型常量来区分两个类，但用类参数也行。在构造函数与析构函数的内部，RTTI的信息用来产生类的名字并显示，类X既用了继承又用了组合来创建一个类，这里构造函数与析构函数调用的顺序很有趣。

这种技术有助于我们理解C++语言是如何工作的。

18.4 引用

RTTI必须能与引用一起工作。指针与引用存在明显不同，因为引用总是由编译器逆向引用，而一个指针的类型或它指向的类型可能要检测。请看下例：

```
class B {
public:
  virtual float f() { return 1.0;}
};
class D : public B { /* ... */ };
B* p = new D;
B& r = *p;
```

typeid()看到的指针类型是基类而不是派生类，而它看到的引用类型则是派生类：

```
typeid(p) == typeid(B*)
typeid(p) != typeid(D*)
typeid(r) == typeid(D)
```

与此相反，指针指向的类型在 typeid()看来是派生类而不是基类，而用一个引用的地址时产生的是基类而不是派生类。

```
typeid(*p) == typeid(D)
typeid(*p) != typeid(B)
typeid(&r) == typeid(B*)
typeid(&r) != typeid(D*)
```

表达式也可以用typeid()运算符，因为它们也有一个类型：

```
typeid(r.f()) == typeid(float)
```

异常

对一个引用完成了一个动态映射，其结果还必须被指定到一个引用上。但如果映射失败则会产生什么呢？因为不能有空的引用，所以这里是抛出一个异常的合适地方。在标准 C++中异常类型为bad-cast，但在下面的例子中，用一个处理块来捕获所有异常：

```
class X {};

mi MI;
d1 & D1 = MI; // upcast to reference
try {
  X& xr = dynamic_cast<X&>(D1);
} catch(...) {
  cout << "dynamic_cast<X&>(D1) failed"
       << endl;
}
```

失败的原因当然是因为D1实际上并不指向一个X对象，如果这里没有抛出一个异常， xr就会没有边界，所有被创建的对象或引用的安全保障都可能被打破。

如果在调用typeid()时试图去除一个空指针的引用，也会引起一个异常，在标准 C++中，这个异常叫bad-typeid:

```
B* bp = 0;
try {
  typeid(*bp); // throws exception
} catch(bad_typeid) {
  cout << "Bad typeid() expression" << endl;
}
```

这里可以在typeid操作之前检查指针是否为空来避免异常的产生（不像上面的那个引用的例子），这是最好的方法。

18.5 多重继承

当然，RTTI机制必须适用于任何复杂的多重继承，包括virtual基类：

```
//: MIRTTI.CPP -- MI & RTTI
#include <iostream.h>
#include <typeinfo.h>

class BB {
public:
 virtual void f() {}
};
class B1 : virtual public BB {};
class B2 : virtual public BB {};
class MI : public B1, public B2 {};

main() {
  BB* bbp = new MI; // Upcast
  // Proper name detection:
  cout << typeid(*bbp).name() << endl;
  // Dynamic_cast works properly:
  MI* mip = dynamic_cast<MI*>(bbp);
  // Can't force old-style cast:
  //! MI* mip2 = (MI*)bbp; // Compile error
}
```

即使只提供一个virtual基类指针，typeid()也能准确地检测出实际对象的名字。用动态映射同样也会工作得很好，但编译器将不允许我们试图用原来的方法强制映射：

```
MI* mip = (MI*)bbp; // compile-time error
```

编译器知道这不可能正确，所以它要求我们用动态映射。

18.6 合理使用RTTI

因为RTTI可以让我们用一个匿名的多态指针来发现类型信息，所以它常常被初学者滥用，因为它可能在虚函数完成之前就有意义了。

对于许多来自过程编程背景的人来说，要他们不把程序组织成为一组 switch语句是非常困难的。他们可能会用 RTTI完成这些，但这样会在代码开发维护阶段丢失多态性的非常重要的价值。C++的意图是：尽可能地使用虚函数，必要时才使用 RTTI。

当然，要想以我们所想的那样使用虚函数，我们必须控制基类的定义，因为随着程序的不断扩大，有时我们可能发现基类并没有我们想要的虚函数，如果基类来自类库或其他由别人控制的来源，就可以用 RTTI作为一种解决办法：我们可以继承一个新类并加上我们的成员函数。在代码的其他地方我们可以检测到我们的新增类型和调用的那个成员函数。这不会破坏多态性和程序逻辑的可扩展性，因为加一个新类并不要求我们寻找 switch语句。当然如果在主程序中增加新的代码时用到了这个新类，我们就必须检测这个特定类型。

把一个特征放在一个基类中可能意味着为了某个特定类的利益，所有从该类派生出的类都保留了一些无意义的虚函数的残留。这使得接口变得不清晰，使那些必须重新定义纯虚函数的人当他们从这个类派生新类时感到很不方便。比方说，假设在第 14章（14.6）节的WINDS.CPP程序中，我们想清除管弦乐队中所有乐器的无用值。一种方法是在基类 instrument

中放一个虚函数ClearSpitvalve()，但这就会引起混乱，因为它暗示percussion和electronic乐器也有无用值。RTTI提供了一个更合理的方法，因为可以把函数放在一个合适的特定类中（这里是wind）。

最后，RTTI有时可以解决效率问题。如果代码用一种好的方法来用多态机制，但结果是这种通用代码对某个对象起反作用，使其运行效率低下。我们可以用 RTTI将这种类型找出来，并写出针对特定情况的代码以提高效率。

回顾垃圾再生器例子

下面是第15章垃圾再生例子（trash recycling）的一个相似的版本，这里我们没有在类层次中建立类信息，而是采用了RTTI：

```
//: RECYCLE2.CPP -- Chapter 14 example w/ RTTI
#include <fstream.h>
#include <stdlib.h>
#include <time.h>
#include <typeinfo.h>
#include "..\14\tstack.h"
ofstream out("recycle2.out");

class trash {
  float Weight;
public:
  trash(float Wt) : Weight(Wt) {}
  virtual float value() const = 0;
  float weight() const { return Weight; }
  virtual ~trash() {}
};

class aluminum : public trash {
  static float val;
public:
  aluminum(float Wt) : trash(Wt) {}
  float value() const { return val; }
  static void value(int newval) {
    val = newval;
  }
};

float aluminum::val = 1.67;
class paper : public trash {
  static float val;
public:
  paper(float Wt) : trash(Wt) {}
  float value() const { return val; }
  static void value(int newval) {
```

```
    val = newval;
  }
};

float paper::val = 0.10;

class glass : public trash {
  static float val;
public:
  glass(float Wt) : trash(Wt) {}
  float value() const { return val; }
  static void value(int newval) {
    val = newval;
  }
};

float glass::val = 0.23;

// Sums up the value of the trash in a bin:
template<class T> void
SumValue(const tstack<T>& bin, ostream& os) {
  tstackIterator<T> tally(bin);
  float val = 0;
  while(tally) {
    val += tally->weight() * tally->value();
    os << "weight of "
       << typeid(*tally.current()).name()
       << " = " << tally->weight() << endl;
    tally++;
  }
  os << "Total value = " << val << endl;
}

main() {
  // Seed the random number generator
  time_t t;
  srand((unsigned)time(&t));

  tstack<trash> bin; // Default to ownership
  // Fill up the trash bin:
  for(int i = 0; i < 30; i++)
    switch(rand() % 3) {
      case 0 :
        bin.push(new aluminum(rand() % 100));
        break;
      case 1 :
```

```
        bin.push(new paper(rand() % 100));
        break;
      case 2 :
        bin.push(new glass(rand() % 100));
        break;
    }
  // Note difference w/ chapter 14: Bins hold
  // exact type of object, not base type:
  tstack<glass> glassbin(0); // No ownership
  tstack<paper> paperbin(0);
  tstack<aluminum> ALbin(0);
  tstackIterator<trash> sorter(bin);
  // Sort the trash:
  while(sorter) {
    aluminum* ap =
      dynamic_cast<aluminum*>(sorter.current());
    paper* pp =
      dynamic_cast<paper*>(sorter.current());
    glass* gp =
      dynamic_cast<glass*>(sorter.current());
    if(ap) ALbin.push(ap);
    if(pp) paperbin.push(pp);
    if(gp) glassbin.push(gp);
    sorter++;
  }
  SumValue(ALbin, out);
  SumValue(paperbin, out);
  SumValue(glassbin, out);
  SumValue(bin, out);
}
```

这个问题的本质是这些垃圾被扔进了一个没有分类的单一的垃圾箱中，所以特定的类型信息被丢失了。但之后特定类型信息必须恢复以便对垃圾准确分类，所以 RTTI被用上了。在第15章中，一个RTTI系统插入到类的继承关系中，但正如我们在这里看到的那样，用 C++预定义的RTTI更方便。

18.7 RTTI的机制及花费

典型的RTTI是通过在VTABLE中放一个额外的指针来实现的。这个指针指向一个描述该特定类型的typeinfo结构（每个新类只产生一个typeinfo的实例），所以typeid()表达式的作用实际上很简单。VPTR用来取typeinfo的指针，然后产生一个结果typeinfo结构的一个引用——这是一个决定性的步骤——我们已经知道它要花多少时间。

对于dynamic_cast<目标*><源指针>，多数情况下是很容易的，先恢复源指针的 RTTI信息再取出目标*的类型RTTI信息，然后调用库中的一个例程判断源指针是否与目标 *相同或者是目标*类型的基类。它可能对返回的指针做了一点小的改动，因为目的指针类可能存在多重继承的情况，而源指针类型并不是派生类的第一个基类。在多重继承时情况会变得复杂些，因为

一个基类在继承层次中可能出现一次以上，并且可能有虚基类。

用于动态映射的库例程必须检查一串长长的基类列表，所以动态映射的开销比 typeid()要大（当然我们得到的信息也不同，这对于我们的问题来说可能很关键），并且这是非确定性的，因为查找一个基类要比查找一个派生类花更多的时间。另外动态映射允许我们比较任何类型，不限于在同一个继承层次中比较两个类。这使得动态映射调用的库例程开销更高了。

18.8 创建我们自己的RTTI

如果编译器还不支持RTTI，可以在类库中很容易地建立自己的RTTI。这是很有意义的事情，因为在人们发现所有的类库实际上都有要用到某种形式的 RTTI之后才在C++引入RTTI。（在异常处理被加入到C++后，人感觉“自由”一些了，因为异常处理要求有关类的准确信息）。

从本质上说，RTTI只要两个函数就行了，一个用来指明类的准确类型的虚函数，一个取得基类的指针并将它向下映射成派生类，这个函数必须产生一个指向更加派生类的指针（我们可能希望也能处理引用）。有许多方法来实现我们自己的RTTI，但都要求每个类有一个唯一的标识符和一个能产生类型信息的虚函数。下例用了一个叫 dynacast()的静态成员函数，它调用一个类型信息函数dynamic_type()，这两个函数都必须在每个新派生类中重新定义：

```
//: SELFRTTI.CPP -- Your own RTTI system
#include "..\14\tstack.h"
#include <iostream.h>
class security {
protected:
  enum { baseID = 1000 };
public:
  virtual int dynamic_type(int ID) {
    if(ID == baseID) return 1;
    return 0;
  }
};

class stock : public security {
protected:
  enum { typeID = baseID + 1 };
public:
  int dynamic_type(int ID) {
    if(ID == typeID) return 1;
    return security::dynamic_type(ID);
  }
  static stock* dynacast(security* s) {
    if(s->dynamic_type(typeID))
      return (stock*)s;
    return 0;
  }
};

class bond : public security {
```

```
protected:
  enum { typeID = baseID + 2 };
public:
  int dynamic_type(int ID) {
    if(ID == typeID) return 1;
    return security::dynamic_type(ID);
  }
  static bond* dynacast(security* s) {
    if(s->dynamic_type(typeID))
      return (bond*)s;
    return 0;
  }
};

class commodity : public security {
protected:
  enum { typeID = baseID + 3};
public:
  int dynamic_type(int ID) {
    if(ID == typeID) return 1;
    return security::dynamic_type(ID);
  }
  static commodity* dynacast(security* s) {
    if(s->dynamic_type(typeID))
      return (commodity*)s;
    return 0;
  }
  void special() {
    cout << "special commodity function\n";
  }
};

class metal : public commodity {
protected:
  enum { typeID = baseID + 4};
public:
  int dynamic_type(int ID) {
    if(ID == typeID) return 1;
    return commodity::dynamic_type(ID);
  }
  static metal* dynacast(security* s) {
    if(s->dynamic_type(typeID))
      return (metal*)s;
    return 0;
  }
```

```
};

main() {
  tstack<security> portfolio;
  portfolio.push(new metal);
  portfolio.push(new commodity);
  portfolio.push(new bond);
  portfolio.push(new stock);
  tstackIterator<security> it(portfolio);
  while(it) {
    commodity* cm =
      commodity::dynacast(it.current());
    if(cm) cm->special();
    else cout << "not a commodity" << endl;
    it++;
  }
  cout << "cast from intermediate pointer:\n";
  security* sp = new metal;
  commodity* cp = commodity::dynacast(sp);
  if(cp) cout << "it's a commodity\n";
  metal* mp = metal::dynacast(sp);
  if(mp) cout << "it's a metal too!\n";
}
```

每个子类必须创建它自己的typeID，重新定义虚函数dynamic_type()来返回这个typeID，并定义一个静态成员调用 dynacast()，它用一个基类指针作为参数（或继承层次中任意层上的一个指针——在这种情况下，指针被简单地向上映射）。

在从security派生出的类中，可以看到每个类都定义了自己的 typeID，并加到baseID中。baseID可以从派生类中直接访问，这一点是关键所在，因为enum必须在编译时计算出值的大小，所以采用内联函数的方法来读一个私有数据成员的方法不会成功。这是一个需要 protected成员的一个典型事例。

enum baseID为所有从security派生出的类建立了一个基本的标识符，这样如果一个 ID值与已有ID值发生冲突，可能改变一下基值就可以改变所有的ID值（因为这个例子中并不比较不同的继承树，所以不可能发生 ID冲突）。在所有的类中，类的ID值都是protected，所以它可以被派生类访问，但终端用户则不能访问它们。

这个例子说明了创建RTTI需要处理哪些事情。我们不仅要确定对象的准确类型，还要能判断这个类是不是从我们要找的类中派生出来的。比如： metal是从commodity派生出来的，commodity有一个叫special()的函数，所以如果有一个metal类的对象，就可以调用special()。如果dynamic_type()只告诉我们这个对象的准确类型，当我们问它一个 metal对象是否是commodity对象时，它会说“不是”，而这实际上是不正确的。所以 RTTI还必须能合理地在继承层次中将一种类型映射到某一中间类型上和准确类型上。

dynacast()函数通过调用虚函数dynamic_type()来确定类型信息。这个函数用一个我们正试图映射到的类的 typeID为参数。它是一个虚函数，所以函数体在对象的准确类型中。每个dynamic_type()函数首先检查传入的 typeID是否与自己的类型匹配，它检查是否与基类匹配，

这只要调用基类的dynamic_type()函数就行了。就像一个循环函数调用，每个dynamic_type()都检查传入的参数是否与自己的ID值相等，如不匹配，它调用基类的dynamic_type()，并将它结果返回。当一直找到继承树的根部时，它将返回零，表示没有匹配的类。

如果dynamic_type()返回1(true)，则指针指向的对象要么就是我们要找的类型，要么是这个类的派生类，然后dynacast()用security指针作参数，并把它映射成想要的类型，如果返回值是false,，dynacast()返回零，表示映射不成功，用这种方法使它看上去就像 C++中的dynamic_cast运算符一样。

C++的动态映射运算符比上面的例子多一项功能：它可以比较两个继承层次中的类型，这两个继承层次可以是完全分开的。这就增加了系统的通用性，使它适用于跨层次体系的类型比较，当然这也增加了系统的复杂性。

现在我们很容易想像出怎样创建一个使用上面方案并允许更容易转换成内置 dynamic_cast运算符的DYNAMIC-CAST宏来。

18.9 新的映射语法

无论什么时候用类型映射，都是在打破类型系统 [1]，这实际上是在告诉编译器，即使知道一个对象的确切类型，还是可以假定认为它是另外一种类型。这本身就是一种很危险的事情，也是一个容易发生错误的地方。

不幸的是，每一种类型映射都是不同的：它是用括号括起来的目标类型的名字。所以如果我们的一段代码不能正确工作，而我们知道应该检查所有的类型映射看它们是否是产生错误的原因。我们怎么保证可以找出所有的类型映射呢？在一个 C程序中无法做到这一点。因为C编译器并不总是要求类型映射（它可以用一个 void指针指向不同的类型而不必强迫使用映射），而映射表现不同，所以我们不知道我们是不是已经找出所有的映射了。

为了解决这个问题， C++用保留字dynamic_cast(本章第一部分的主题)、const_cast、static_cast和reinterpret_cast来提供了一个统一的类型映射语法。当需要进行动态映射时，这就提供了一个解决问题的可能。这意味着那些已有的映射语法已经被重载得太多了，不能再支持任何其他的功能了。

通过使用这些映射来代替原有的（ newtype）语法，我们可以在任何程序中很容易地找出所有的映射。为了支持已有的代码，大多数编译器都可以产生不同级别的错误或警告，并可由用户对错误或警告产生选择打开或关闭。如果把新类型映射的全部错误打开的话，就可以确保我们找出项目中所有的类型映射，这使得查找错误变得很容易。

下表指出了四个不同形式的映射的含义：

static_cast	为了“行为良好”和“行为较好”而使用的映射，包括一些我们可能现在不用的映射（如向上映射和自动类型转换）
const_cast	用于映射常量和变量（const和volatile）
dynamic_cast	为了安全类型的向下映射（本章前面已经介绍）
reinterpret_cast	为了映射到一个完全不同的意思。这个关键词在我们需要把类型映射回原有类型时要用到它。我们映射到的类型仅仅是为了故弄玄虚和其他目的。这是所有映射中最危险的

三个新映射将在后面小节中完整地介绍。

[1] 参看Josée Lajoie“The new cast notation and the bool data type” ，C++报告，1994年9月，pp.46-51.

18.9.1 static_cast

static_cast可以用于所有良定义转换。这些包括“安全”转换与次安全转换，“安全”转换是编译器允许我们不用映射就能完成的转换。次安全转换也是良定义的。由 static_cast覆盖的变换的类型包括典型的无映射变换、窄化变换（丢失信息）、用void*的强制变换、隐式类型变换和类层次的静态导航。

```
//: STATCAST.CPP -- Examples of static_cast

class base { /* ... */ };
class derived : public base {
public:
  // ...
  // Automatic type conversion:
  operator int() { return 1; }
};

void func(int) {}

class other {};

main() {
  int i = 0x7fff; // Max pos value = 32767
  long l;
  float f;
  //(1) typical castless conversions:
  l = i;
  f = i;
  // Also works:
  l = static_cast<long>(i);
  f = static_cast<float>(i);

  //(2) narrowing conversions:
  i = l; // May lose digits
  i = f; // May lose info
 // Says "I know," eliminates warnings:
 i = static_cast<int>(l);
 i = static_cast<int>(f);
 char c = static_cast<char>(i);

 //(3) forcing a conversion from void* :
 void* vp = &i;
 // Old way produces a dangerous conversion:
 float* fp = (float*)vp;
 // The new way is equally dangerous:
 fp = static_cast<float*>(vp);
```

```
  //(4) implicit type conversions, normally
  // Performed by the compiler:
  derived d;
  base* bp = &d; // Upcast: normal and OK
  bp = static_cast<base*>(&d); // More explicit
  int x = d; // Automatic type conversion
  x = static_cast<int>(d); // More explicit
  func(d); // Automatic type conversion
  func(static_cast<int>(d)); // More explicit

  //(5) Static Navigation of class hierarchies:
  derived* dp = static_cast<derived*>(bp);
  // ONLY an efficiency hack. dynamic_cast is
  // Always safer. However:
  // Other* op = static_cast<other*>(bp);
  // Conveniently gives an error message, while
  other* op2 = (other*)bp;
  // Does not.
}
```

在第（1）段中，我们看到了在C语言中使用过的各种类型的转换，用映射的或不用映射的。从一个int到一个long或float当然不成问题，因为后者可以存放 int包含的全部值，可以用一个static_cast来使这些转换变得醒目一些，虽然并不是必须要这样做。

在第（2）段中，我们可以看到转换回原有类型的方法。这里可能丢失部分数据，因为一个int没有一个long或float那么“宽”。因此这些被称为“窄化转换”，编译器仍可以完成这些转换，但会给我们一个警告信息。可以去掉这些警告，并用一个映射指出我们确实需要映射。

在C++中，从void*向外赋值是不允许的（这与C中不同），请看第（3）段。这样做是很危险的，它要求程序员清楚地知道他正在干什么。当我们查找错误时， static_cast比老式的标准映射更容易定位。

第（4）段显示了几种隐式类型转换，这些通常是由编译器自动完成的。它们是自动的和不要求映射的，用static_cast会使它变得醒目，以便于我们以后需要查找它或明确它们的含义。

如果在一个类层次中没有虚函数或者如果我们有其他允许我们安全地向下映射的信息，则用静态向下映射比 dynamic_cast稍微快一些，就像在第（ 5）段中显示的那样。另外，static_cast不允许我们映射到类层次之外，就像传统的映射一样，所以它更安全。然而静态地导航类层次总是冒险的，因此我们应该用dynamic_cast，除非特殊情况。

18.9.2 const_cast

如果想把一个 const 转换为非const，或把一个volatile转换成一个非 volatile，就要用到const_cast。这是可以用const_cast的唯一转换。如果还有其他的转换牵涉进来，它必须分开来指定，否则会有一个编译错误。

```
//: CONSTCST.CPP -- Const casts

main() {
```

```
  const int i = 0;
  int* j = (int*)&i; // Deprecated form
  j  = const_cast<int*>(&i); // Preferred
  // Can't do simultaneous additional casting:
//! long* l = const_cast<long*>(&i); // Error
  volatile int k = 0;
  int* u = const_cast<int*>(&k);
}

class X {
  int i;
// mutable int i; // a better approach
public:
  void f() const {
    // Casting away const-ness:
    (const_cast<X*>(this))->i = 1;
  }
};
```

如果用了一个const对象的地址创建一个指针指向一个 const，在没有一个映射时不能把它赋给一个非const的指针中。老式风格的映射可以完成这个，但使用 const_cast更合适。这同样适用于volatile。

如果想在一个const成员函数内部改变一个类成员，传统的方法就是用（ X*）this映射掉常量性质。现在也可以使用较好的 const_cast来映射掉常量性质，但一个更好的方法是使那些特殊的数据成员成为 mutable，这样在类定义时它更清楚，不会在成员函数定义中被隐藏掉，而且这些成员可以在const成员函数中改变。

18.9.3 reinterpret_cast

这是一种不太安全的类型映射机制，也是最容易引起错误的一种。一般情况下，编译器都包含了一组开关，允许我们强制使用 const_cast和reinterpret_cast，它们可以定位那些不安全的类型映射。

reinterpret_cast假设一个对象仅仅是一个比特模式，它可以被当作完全不同的对象对待（为了某种模糊的目的）。这是低层处理，在C中已经很不好了。实际上在我们用它做别的事情之前，总是要用reinterpret_cast将其映射回原来的类型。

```
//: REINTERP.CPP -- Reinterpret_cast
// Example depends on VPTR location,
// Which may differ between compilers.
#include <string.h>
#include <fstream.h>
ofstream out("reinterp.out");

class X {
  enum { sz = 5 };
  int a[sz];
```

```
public:
  X() { memset(a, 0, sz * sizeof(int)); }
  virtual void f() {}
  // Size of all the data members:
  int membsize() { return sizeof(a); }
  friend ostream&
    operator<<(ostream& os, const X& x) {
      for(int i = 0; i < sz; i++)
        os << x.a[i] << ' ';
      return os;
  }
};

main() {
  X x;
  out << x << endl; // Initialized to zeroes
  int* xp = reinterpret_cast<int*>(&x);
  xp[1] = 47;
  out << x << endl; // Oops!

  X x2;
  const vptr_size = sizeof(X) - x2.membsize();
  long l = reinterpret_cast<long>(&x2);
  // *IF* the VPTR is first in the object:
  l += vptr_size; // Move past VPTR
  xp = reinterpret_cast<int*>(l);
  xp[1] = 47;
  out << x2 << endl;
}
```

类X包含一些数据和一个虚函数，在main()中，一个X的对象被打印出以显示出它已被初始化为零了，然后它的地址用reinterpret_cast映射为一个int*，假设它是一个int*，这个对象被索引成像一个数组，并且成员1被置为47（理论上），但在这里输出结果[1]却是：

```
0 0  0  0  0
47 0  0  0  0
```

很明显，认为对象的第一个数据存放在对象的起始地址处这一假定是不安全的。事实上，这个编译器把VPTR放在对象的开始处，所以如果用xp[0]而不是用xp[1],就会使VPTR变得毫无价值。

为了更正这个错误，可以用对象的大小减去数据成员的大小算出 VPTR的大小，然后对象的地址被映射为一个 long型(用reinterpret_cast)。假定VPTR是放在对象的开始处的，这样，实际数据的开始地址就被计算出来了。结果数字映被射回 int*，现在索引值可以产生想要的结果了：

```
0  47  0  0  0
```

[1] 对于特定的编译器，结果可能不同。

当然这种方法不值得推荐，而且可移植性差。这是一个 reinterpret_cast指示器能做的事情之一，但当我们决定我们必须要用它时，它是可用的。

18.10 小结

RTTI是一个很方便的额外特征，就像蛋糕上加了一层糖衣。虽然一般都是把一个指针向上映射为一个基类指针，然后使用基类的接口（通过虚函数），但是偶尔需要知道一个基类指针指向的对象的确切类型来提高程序的效率，这时如果我们束手无策，就可使用 RTTI。因为基于虚函数的RTTI已经出现在几乎所有的类库中，所以这是一个非常有用的特征，因为它意味着：

1) 我们并不需要把它建在其他类库中。
2) 我们不用担心它是否将建在其他库中。
3) 在继承过程中我们不需要有额外的编程花费用来管理 RTTI配置。
4) 语法是一致的，我们不需要为每个新库来重新配置。

因为RTTI使用很方便，像C++的多数特征一样，所以它可能被滥用，包括那些天真的或有决心的程序员。最常见的滥用可能来自那些不理解虚函数的程序员，他们用 RTTI去做类型检查编码。C++的哲学似乎是提供强有力的工具并维护类型的完整和防止类型的违规，但我们如果有意滥用某一个特征的话，没有什么可以阻止我们。有时走点小弯路是获取经验的最快途径。

新的类型映射语法在调试阶段对我们有很大的帮助，因为这种类型映射在我们的类型系统中开了一个小洞，并允许错误流进去。而这种新的语法使我们更容易定位这些错误入口通道。

18.11 练习

1. 用RTTI帮助程序调试，即打印一个使用了 typeid()的模板的确切名称。用不同的类型将这个模板实例化，然后看看结果是什么。

2. 用RTTI实现本章前面讲的TurnColorIfYouAreA()函数。

3. 将第14章的WIND5.CPP拷贝到一个新的位置，然后修改其中的 instrument的层次。在wind类中加一个虚函数ClearSpitValve()并在所有的wind的派生类中重新定义它。让 tstash的一个实例拥有一些instrument指针，把用new创建的各类instrument对象赋给它们。现在用RTTI巡视这个包容器，在所有的对象中找类 wind或它的派生类的对象，为这些对象调用ClearSpitValve()函数。注意，如果instrument基类中已经包含了ClearSpitValve()函数，它可能会引起混乱。

附录A 其他性能

在写这本书时，C++的标准还没有制定出来。虽然实际上所有这些特征最终都会被加入到这个标准中，但有些并没有在所有的编译器中出现。这个附录中简单地介绍了一些其他特征，这些应该在编译器中（或在编译器的未来版本中）去查找。

A.1 bool、true、false

实际上每个人都在用布尔形变量，而且每个人定义它们的方式都不相同[1]，有些人用枚举，另一些人用typedef。 typedef是一个特殊的问题，因为不能重载它（一个typedef对于一个int还是一个int），也不能用它初始化一个唯一的模板。

在标准库中，可能为bool类型创建了一个类，但这也不能很好地工作，因为我们只能有了一个自动类型转换运算符，而没有解决重载问题。

对于这样一个有用的类型，最好的方法是把它建在语言内部。一个 bool型有两种状态，它们由内部常量true（它转化成一个整数1）和false(它转化成一个整数0)表示，这三个名字都是关键词，另外对部分语言成分作了修改：

成 分	bool型的用法
&&‖！	取bool型参数，返回bool值
< > <= >= == !=	产生bool型结果
if ,for while,do	条件表达式转换为一个bool值
?:	第一个操作数转换为bool值

因为已有的许多代码常用一个int表示一个标志，编译器将一个int隐式转换成一个bool型。理想情况下，编译器将给我们一个警告，以建议我们改正这种情况。

一种俗话称“低劣的编程风格”的情况是用 ++来把一个标志置为true。这是允许的，但不赞成这样做，这意味着在将来某个时候它会变成不合法的。这个问题与 enum的增运算一样：我们正在做从bool 型转化成int型的隐式类型转换，增加这个值（可能越出一般的 bool值0~1的范围），然后隐式地映射回来。

在必要时指针也可以自动转换成bool型的。

A.2 新的包含格式

随着C++的不断发展，不同的编译器开发商选择了不同的文件扩展名。另外，不同的操作系统对文件名有不同的限制，特别是在文件名的长度上。为了适应各种不同的情况， C++标准采纳了一个新的格式，它允许文件名突破那很不好的八个字符的限制，并且取消了扩展名。比如，包含IOSTREAM.H就变成了：

```
# include  <iostream>
```

[1] 见“Josée Lajoie，"The new cast notation and the bool data type”C++报告，1994年9月。

解释器按特定的编译器和操作系统去实现文件的包含，必要时缩短文件名并加上一个扩展名。如果我们想在编译器开发商支持这一特性之前使用这一风格的包含文件，也可以把开发商提供给我们的头文件拷贝到不带扩展名的文件中去。

所有从标准C中继承来的库在我们包含它们时还是用 .h作为扩展名，这使读者很容易从我们使用的C++库中识别出C库。

A.3 标准C++库

标准的C++不仅包含了全部的标准C库（做了一点小的增补和改动，以支持安全类型），而且还增加了一些它自己的库。这些库比标准的 C库功能更强，从中获得的益处与从 C向C++转变获得的益处类似。对这些库的最好的参考文献就是标准本身（写这本书时还只能得到非正式版），可以在 Internet或BBS上找到它们。

输入输出流库已在本书第6章做了介绍，下面简要介绍一下C++中其他常用的库：

语言支持：包括继承到本语言中来的成分，像 <climits>和<cfloat>中的实现限制；<new>中的动态内存声明，例如bad_alloc（当我们超出内存范围时抛出的异常）和 set_new_handler；用于RTTI的<typeinfo>头文件和声明了terminate()和unexpected() 函数的<exception>头文件。

诊断库：一些组件，C++程序能够用以发现和报告错误。<stdexcept>头文件声明了标准异常类，<cassert>与C中ASSERT.H的作用相同。

通用实用库：这些组件被标准C++库的其他部分使用，我们也可以在我们的程序中使用它们。包括运算符!=、>、<=和>=（为防止多余的定义）的模板化版本、带 tuple模板函数的pair模板类、支持STL（在本附录的下一节中介绍）的一套函数对象和与 STL一起使用的能使我们可以很容易地修改存储分配机制的内存分配函数。

字符串库：字符串类可能是我们曾经见过的最完整的字符串处理工具。我们在 C中用了数行代码来完成的工作都可以用字符串类中的一个成员函数来代替，包括 append()、assign()、insert()、remove()、replace()、resize()、copy()、find()、rfind（）、find_first_of()、find_last_of()、find_first_not_of()、find_last_not_of()、substr()和compare()。此外还重载了运算符=、+=和[]使这些运算更直观。另外有一个“宽字符”的 wstring类，用来支持国际字符集。string和wstring (在<string>中声明，不要和C中的STRING.H弄混)都是从一个叫basic_string的通用模板类中产生的。注意字符串类和输入输出流已经无缝地结合在一起了，所以我们甚至无需用strstream了（也不用担心第6章中描述的有关内存管理了）。

本地化库：这个库用来调整字符集以使我们的程序能在不同国家使用，包括货币、数字、日期、时间等等。

包容器库：这包括标准的模板库（在本附录的下一节中介绍）以及 <bits>和<bitstring>中的bits类和bit_string类。bits和bit_string都更完整地实现了第5章中介绍的位向量的概念。bits模板产生一个固定大小的位数组，可以用所有的位运算符对其运算，同时包含 set() 、reset() 、count()、length()、test()、any() 和none()等成员函数。另外还有几个转换运算符 to_ushort()、to_ulong()和to_string()。

bit_string则相反，它是一个动态长度的位数组，它不仅包含 bits同样的运算符，还包含一些其他运算使它看上去像string类。bits和bit_string在位的权重上有一个根本的区别：对于 bits，最右的位（第 0位）是最不重要的位，但在 bit_string中，最右的位是最有意义的位。 bits和bit_string之间不能互相转换。我们可以把 bits用于一组节省空间的开关变量，而用 bit_string来管理二进制值的数组（如像表素）。

循环子库：包括用于STL的工具（下一节中介绍）、流及流缓冲区。

算法库：这些都是一些模板函数，用循环子来完成 STL包容器上的运算，它包括adjacent_find、prev_permutation、binary_search、push_heap、copy、random_shuffle、copy_backward、remove、count、remove_copy、count_if、remove_copy_if、equal,、remove_if、equal_range、replace、fill、replace_copy、fill_n、replace_copy_if、find、replace_if、find_if、reverse、for_each、reverse_copy、generate、rotate、generate_n、rotate_copy、includes、search、inplace_merge、set_difference、lexicographical_compare、set_intersection、lower_bound、set_symmetric_difference、make_heap、set_union、max、sort、max_element、sort_heap、merge、stable_partition、min、stable_sort、min_element、swap、mismatch、swap_ranges、next_permutation、transform、nth_element、unique、partial_sort、unique_copy、partial_sort_copy、upper_bound和partition。

数字库：这个库的目的是允许编译器的实现者在用数字运算时充分利用低层机器结构。这样更高层的数字库可以写到这个数字库中，以产生更高效的代码，而不必为每种可能的机器都编写代码。这个数字库也包括复杂的数字类（这些在C++的第一个版本中是以例子出现的，现在已经是这个库的一部分了），这些类以float、double和long double的形式出现。

A.4 标准模板库（STL）

第15章提供的模板说明了C++中模板的强大功能和灵活性，但它们并不仅仅是作为一般目的的工具，虽然它们确实可以解决一些问题。C++的STL [1]是一个功能强大的库，它可以满足我们对包容器和算法的巨大需求，而且是一种完全可移植的方式。这意味着程序不仅可以很容易地移植到其他平台上，而且我们的知识本身并不依赖某个特定编译器开发商提供的库。因此在寻找特定开发商的解决方案之前应该首先在STL中寻找包容器和算法。

软件设计有一个基本原则：就是所有的问题都可以通过引进一个间接层来简化。这种简化在STL中是用循环子来完成的，它在尽可能少地知道某种数据结构的情况下完成对这一结构的运算，因此它使得数据结构与STL相独立，这意味着所有运用于内部类型上的运算也可以在用户定义的类型上完成，反之亦然。如果我们学会了这个库，就可以运用于每种情况。

这种独立性的缺点是我们得花时间去熟悉STL中处理事情的方法。当然，STL用的是统一的模式，所以我们一旦适应了它，从一个STL工具到另一个工具，就不会有什么变化。。

请看下面用STL的set类的一个例子，一个简单的set可以创建与int一起工作：

```
//: INTSET.CPP -- Simple use of STL set
#include <set.h>

void main() {
  set<int, less<int> > intset;
  for(int i = 0; i < 25; i++)
    for(int j = 0; j < 10; j++)
      // Try to insert multiple copies:
      intset.insert(j);
  // Print to output:
  copy(intset.begin(), intset.end(),
```

[1] 由Alexander Stepanov和Meng Lee在惠普促成为C++标准。

```
        ostream_iterator<int>(cout, "\n"));
}
```

set的第一个参数是包含在这个集合中的类型，第二个参数产生一个运算用于在集合中插入元素（用insert()函数）。这个集合将允许不同的关键字插入到集合中。

copy()函数显示了循环子的使用。set的成员函数begin()和end()产生循环子作为它们的返回值。copy()函数就用这些指针作为它运算的开始点和结束点，并在由它们产生的边界之间简单移动并把元素拷贝到第三个参数上，这第三个参数也是一个循环子，但它是为输入输出流而产生的特殊类型。这就将一个int对象放到了cout上并用一个新行来分割它们。

copy()函数在一个流上打印是不受限制的。它实际上可用于任何场合：它只需要三个循环子来交流数据即可。所有的算法都遵从copy()的这种形式，并简单地管理循环子（这就是额外的间接层）。

现在来看看INTSET.CPP的形式，并改写它来解决第15章中出现的协调性问题，这种解决方法相当简单。

```
//: CONCORD.CPP -- Concordance with STL
#include <set.h>
#include <fstream.h>
#include "..\allege.h"
#include "..\14\sstring.h"
const char* delimiters =
  " \t;()\"<>:{}[]+-=&*#.,/\\"
  "0123456789";

main(int argc, char* argv[]) {
  allege(argc == 2, "usage: concord filename");
  ifstream in(argv[1]);
  allegefile(in);
  set<Hstring, less<Hstring> > concordance;
  const sz = 1024;
  char buf[sz];
  while(in.getline(buf,sz)) {
    // Capture individual words:
    char* s = strtok(buf, delimiters);
    while(s) {
      concordance.insert(s); // Auto type conv.
      s = strtok(0, delimiters);
    }
  }
  // output results:
  copy(concordance.begin(), concordance.end(),
       ostream_iterator<Hstring>(cout, "\n"));
}
```

这里真正的唯一的区别在于用Hstring而不是int。被插入的词是从一个文件中读入的并用strtok()函数作为标志。其他的和INTSET.CPP中完全一样。更方便的是，这种协调性是自动排

序的。

STL中包含的其他类如下：

包 容 器	使 用
vector	随机访问循环，在尾部插入 /删除运算的时间相同。在中间插入 /删除运算将花线性时间，也有用于 bool型的规范说明
list	双向循环和在任何地方顺序插入 /删除运算有一致时间
deque	随机访问循环，在头部、尾部插入 /删除运算的时间相同。在中间插入/删除运算将花线性时间
stack	我们可以选择 vector、list或deque作为模板参数来创建它
queue	可以用list或 deque从模板中创建
priority_queue	随机访问循环，提供 top(),push()和pop()运算，可以从 vector或deque模板中产生
multiset	一个允许有相同关键词的集合
map	一个带唯一关键词的一个联合包容器
multimap	一个允许有相同关键词的联合包容器，像一个哈希 (hash)表

本节只是简单地介绍了STL的功能，我们花点时间来学习STL是值得的。

A.5 asm关键字

这个关键字允许我们在C++程序中用汇编代码来控制我们的硬件。通常可以在汇编代码中引用C++变量，这意味着可以很容易地和我们的 C++代码通信，并且只用在必须高效运行或运用特殊的处理器指令的情况下才使用汇编代码。汇编语言的确切语法是依赖特定编译器的，这可以在编译器文档中找到。

A.6 明确的运算符

这些是位运算和逻辑运算运算符。美国以外地区的键盘上可能没有 &、|、……之类的键，只好使用C中可怕的特殊符号，这不仅难以打印，而且阅读起来非常难懂。这在 C++中用了一些新的关键词来代替。

关 键 词	含 义
and	&&(逻辑与)
or	|| （逻辑或）
not	!（逻辑反）
not_eq	!=（逻辑不等于）
bitand	&（按位与）
and_eq	&=（按位与赋值）
bit or	|（按位或）
or_eq	|=（按位或赋值）
xor	^（按位异或）
xor_eq	^=（按位异或赋值）
compl	~（1的补）

附录B 编 程 准 则

这个附录[1]收集了C++编程的一些建议，它们是我在教学和实践过程中收集而成的，当然还有：

从朋友那儿得到的忠告，包括 Dan Saks（与Tom Plum合写了“ C++ Programming Guidelines ”，Plum Hall 1991）、Scott Meyers(“ Effective C++ ”一书的作者， Addison-Wesley,1992)和Rob Murray（“ C++ strategies & Tactics ”的作者， Addision-Wesley,1993），有许多条目都是从这本书上摘录下来的。

1. 不要用C++主动重写我们已有的C代码，除非我们需要对它的功能做较大的调整，（也就是说，不破不立）。用C++重新编译是很有价值的，因为这可以发现隐藏的错误。把一段运行得很好的C代码用C++重写可能是在浪费时间，除非 C++的版本以类的形式提供许多重用的机会。

2. 要区别类的创建者和类的使用者（客户程序员）。类的使用者才是“ 顾客 ”，他们并不需要或许也不想知道类的内部是怎样运作的。类的创建者必须是设计类和编写类的专家，以使得被创建的类可以被最没有经验的程序员使用，而且在应用程序中工作良好。库只是在透明的情况下才会容易使用。

3. 当我们创建一个类时，要尽可能用有意义的名字来命名类。我们的目标应该是使用户接口要领简单。可以用函数重载和缺省参数来创建一个清楚、易用的接口。

4. 数据隐藏允许我们（类的创建者）将来在不破坏用户代码（代码使用了该类）的情况下随心所欲地修改代码。为实现这一点，应把对象的成员尽可能定义为 private，而只让接口部分为public，而且总是使用函数而不是数据。只有在迫不得已时才让数据为 public。如果类的使用者不需要调用某个函数，就让这个函数成为 private。如果类的一部分要让派生类可见，就定义成protected，并提供一个函数接口而不是直接暴露数据，这样，实现部分的改变将对派生类产生最小的影响。

5. 不要陷入分析瘫痪之中。有些东西只有在编程时才能学到并使各种系统正常。 C++有内建的防火墙，让它们为我们服务。在类或一组类中的错误不会破坏整个系统的完整性。

6. 我们的分析和设计至少要在系统中创建类、它们的公共接口、它们与其他类的关系、特殊的基类。如果我们的方法产生的东西比这些更多，就应当问问自己，是不是所有的成分在程序的整个生命期中都是有价值的，如果不是，将会增加我们对它们的维护开销。开发小组的人都认为不应该维护对他们的产品没有用的东西。许多设计方法并不大奏效，这是事实。

7. 记住软件工程的基本原则：所有的问题都可以通过引进一个额外的间接层来简化(Andrew Koenig向我解释了这一点)。这是抽象方法的基础，而抽象是面向对象编程的首要特征。

8. 使类尽可能地原子化。也就是每个类有一个单一、清楚的目的。如果我们的类或我们设计的系统过于复杂，就应当将所有复杂的类分解成多个简单的类。

9. 从设计的角度，寻找并区分那些变化和不变的成分。也就是在系统中寻找那些修改时不

[1] 增加这个附录是Andrew Binstock建议的，他是《Unix Review》的主编，兼撰稿人。

需要重新设计的成分，把它们封装到一个类中。

10. 注意不同点。两个语义上不同的对象可能有同样的操作或反应，自然就会试着把一个作为另一个的子类以便利用继承性的好处。这就叫差异，但并没有充分的理由来强制这种并不存在的父子关系。一个好的解决办法是产生一个共同的父类：它包含两个子类——这可能要多占一点空间，但我们可以从继承中获益，并且可能对这种自然语言的解有一个重要发现。

11. 注意在继承过程中的限制。最清晰的设计是向被继承者加入新的功能，而如果在继承过程删除了原有功能，而不是加入新功能，那这个设计就值得怀疑了。但这也不是绝对的，如果我们正在与一个老的类库打交道，对已有的类在子类中进行限制可能更有效，而不必重建一套类层次来使我们的新类适应新的应用。

12. 不要用子类去扩展基类的功能。如果一个类接口部分很关键的话，应当把它放在基类中，而不是在继承时加入。如果我们正在用继承来添加成员函数，我们可能应该重新考虑我们的设计。

13. 一个类一开始时接口部分应尽可能小而精。在类使用过程中，我们会发现需要扩展类的接口。然而一个类一旦投入使用，我们要想减少接口部分，就会影响那些使用了该类的代码，但如果我们需要增加函数则不会有影响，一切正常，只需重新编译一下即可。但即使用新的成员函数取代了原来的功能，也不要去改正原有接口（如果我们愿意的话，可以在低层将两个函数合并）。如果我们需要对一个已有的函数增加参数，我们可以让原来的参数保持不变，把所有新参数作为缺省参数，这样不会妨碍对该函数已有的调用。

14. 大声朗读我们的类，确保它们是合理的。读基类时用“ is-a ”，读成员对象时用“ has-a ”。

15. 在决定是用继承还是用组合时，问问自己是不是需要向上映射到基类。如果不需要，就用组合（成员对象）而不用继承。这样可以减少多重继承的可能。如果我们选择继承，用户会认为他们被假设向上映射。

16. 有时我们为了访问基类中的 protected成员而采用继承。这可能导致一个可察觉的对多重继承的需求。如果我们不需要向上映射，首先导出一个新类来完成保护成员的访问，然后把这个新类作为一个成员对象，放在需要用到它的所有对象中去。

17. 一个典型的基类仅仅是它的派生类的一个接口。当我们创建一个基类时，缺省情况下让成员函数都成为纯虚函数。析构函数也可以是纯虚函数（强制派生类对它重新定义），但记住要给析构函数一个函数体，因为继承关系中所有的析构函数总是被调用。

18. 当我们在类中放一个虚函数时，让这个类的所有函数都成为虚函数，并在类中定义一个虚析构函数。当我们要求高效时再把 virtual关键词去掉，这种方法防止了接口的行为出格。

19. 用数据成员表示值的变化，用虚函数表示行为的变化。如果我们发现一个类中有 几个状态变量和几个成员函数，而成员函数在这些变量的作用下改变行为，我们可能要重新设计它，用子类和虚函数来区分这种不同的作用。

20. 如果我们必须做一些不可移植的事，对这种服务做一个抽象并将它定位在一个类的内部，这个额外的间接层可防止这种不可移植性影响我们的整个程序。

21. 尽量不用多重继承。这可帮我们摆脱困境，尤其是修复我们无法控制的类的接口。除非我们是一个经验相当丰富的程序员，否则不要在系统中设计多重继承。

22. 不要用私有继承。虽然 C++中可以有私有继承，而且似乎在某些场合下很有用，但它和运行时类型识别一起使用时，常常引起语义的模棱两可。我们可以用一个私有成员对象来代替私有继承。

23. 运算符重载仅仅是“ 语法糖 ”：另一种函数调用方法。如果重载一个运算符不会使类的

接口更清楚、更易于使用，就不要重载它。一个类只创建一个自动类型转换运算符，一般情况下，重载运算符应遵循第11章介绍的原则和格式。

24. 首先保证程序能运行，然后再考虑优化。特别是，不要急于写内联函数、使一些函数为非虚函数或者紧缩代码以提高效率。这些在我们开始构建系统时都不用考虑。我们开始的目标应该是证明设计的正确性，除非设计要求一定的效率。

25. 不要让编译器来为我们产生构造函数、析构函数或“ = ”运算符。这些是训练我们的机会。类的设计者应该明确地说出类应该做什么，并完全控制这个类。如果我们不想要拷贝构造函数或“ = ”运算符，就把它们声明为私有的。记住，只要我们产生了任何构造函数，就防止了缺省构造函数被生成。

26. 如果我们的类中包含指针，我们必须产生拷贝构造函数、“ = ”运算符和析构函数，以使类运行正常。

27. 为了减少大项目开发过程中的重复编译，应使用第3章介绍的类句柄/Cheshire cat技术，只有需要提高运行效率时才把它去掉。

28. 避免用预处理器。可以用常量来代替值，用内联函数代替宏。

29. 保持范围尽可能的小，这样我们的对象的可见性和生命期也就尽可能的小。这就减少了错用对象和隐藏难以发现的错误的可能性。比方说，假设我们有一个包容器和一段扫描这个包容器的代码，如果我们拷贝这些代码来用一个新的包容器，我们可能无意间用原有的包容器的大小作为新包容器的边界。如果原来的这个包容器超过了这个范围，就会引起一个编译错误。

30. 避免使用全局变量。尽可能把数据放在类中。全局函数存在的可能性要比全局变量大，虽然我们后来发现一个全局函数作为一个类的静态成员更合适。

31. 如果我们需要声明一个来自库中的类或函数，应该用包含一个头文件的方法。比如，如果我们想创建一个函数来写到 ostream中，不要用一个不完全类型指定的方法自己来声明ostream，如

```
class ostream；
```

这样做会使我们的代码变得很脆弱（比如说 ostream实际上可能是一个typedef）。我们可以用头文件的形式，例如：

```
#include <iostream.h>
```

当创建我们自己的类时，如果一个库很大，应提供给用户一个头文件的简写形式，文件中包含有不完全的类型说明（这就是类型名声明），这是对于只需要用到指针的情况（它可以提高编译速度）。

32. 当选择重载运算符的返回值类型时候，要一起考虑串连表达式：当定义运算符“ = ”时应记住x=x。对左值返回一个拷贝或一个引用（返回 *this），所以它可以用在串连表达式（A=B=C）中。

33. 当写一个函数时，我们的第一选择是用const引用来传递参数。只要我们不需要修改正在被传递进入的对象，这种方式是最好的。因为它有着传值方式的简单，但不需要费时的构造和析构来产生局部对象，而这在传值方式时是不可避免的。通常我们在设计和构建我们的系统时不用注意效率问题，但养成这种习惯仍是件好事。

34. 当心临时变量。当调整完成时，要注意临时创建的对象，尤其是用运算符重载时。如果我们的构造函数和析构函数很复杂，创建和销毁临时对象就很费时。当从一个函数返回一个值时，总是应在return语句

```
return foo(i,j);
```

中调用构造函数来“就地”产生一个对象。这优于

```
foo  x(i,j);
return  x;
```

前一个返回语句避免了拷贝构造函数和析构函数的调用。

35. 当产生构造函数时，要考虑到异常，在最好的情况下，构造函数不需要引起一个异常，另一种较好的情况：类将只从健壮的类被组合和继承，所以当一个异常产生时它会自动清除它自己。如果我们必须使用一个裸指针，我们应该负责捕获自己的异常，然后在我们的构造函数中释放所有异常出现之前指针指向的资源。如果一个构造函数无法避免失败，最好的方法是抛出一个异常。

36. 在我们的构造函数中只做一些最必要的事情，这不仅使构造函数的调用有较低的时间花费（这中间有许多可能不受我们控制），而且我们的构造函数更少地抛出异常和引起的问题。

37. 析构函数的作用是释放在对象的整个生命期内分配的所有资源，而不仅仅是在创建期间。

38. 使用异常层次，从标准C++异常层次中继承并嵌套，作为能抛出这个异常的类中的一个公共类。捕获异常的人然后可以确定异常的类型。如果我们加上一个新的派生异常，客户代码还是通过基类来捕获这个异常。

39. 用值来抛出异常，用引用来捕获异常。让异常处理机制处理内存管理。如果我们抛出一个指向在堆上产生的异常的指针，则异常处理器必须知道怎样破坏这个异常，这不是一种好的搭配。如果我们用值来捕获异常，我们需要额外的构造和析构，更糟的是，我们的异常对象的派生部分可能在以值向上映射时被切片。

40. 除非确有必要，否则不要写自己的类模板。先查看一个标准模板库，然后查问创建特殊工具的开发商。当我们熟悉了这些产品后，我们就可大大提高我们的生产效率。

41. 当创建模板时，留心那些不带类型的代码并把它们放在非模板的基类中，以防不必要的代码膨胀。用继承或组合，我们可以产生自己的模板，模板中包含的大量代码都是类型有关的，因此也是必要的。

42. 不要用STDIO.H中的函数，例如printf()。学会用输入输出流来代替，它们是安全类型和可扩展类型，而且功能也更强。我们在这上面花费的时间肯定不会白费（参看第6章）。一般情况下都要尽可能用C++中的库而不要用C库。

43. 不要用C的内部数据类型，虽然C++为了向后兼容仍然支持它们，但它们不像C++的类那样强壮，所以这会增加我们查找错误的时间。

44. 无论何时，如果我们用一个内部数据类型作为一个全局或自动变量，在我们可以初始化它们之前不要定义它们。每一行定义一个变量，并同时对它初始化。当定义指针时，把‘*’紧靠在类型的名字一边。如果我们每个变量占一行，我们就可以很安全地定义它们。这种风格也使读者更易于理解。

45. 保证初始化出现在我们的代码的所有方面。在构造函数初始化表达式表中完成所有成员的初始化，甚至包括内部数据类型（用伪构造函数调用）。用任何簿记技术来保证没有未初始化的对象在我们的系统中运行。在初始化子对象时用构造函数初始化表达式表常常更有效，否则调用缺省构造函数，并且我们最终调用其他成员函数，也可能是运算符“=”，为了得到我们想要的初始化。

46. 不要用“foo a=b;”的形式来定义一个对象。这是常常引起混乱的原因。因为它调用构

造函数来代替运算符“ = ”。为了清楚起见，可以用“ foo a(b); ”来代替。这个语句结果是一样的，但不会引起混乱。

47. 使用C++中新的类型映射。一个映射践踏了正常的类型系统，它往往是潜在的错误点。通过把C中一个映射负责一切的情况改成多种表达清楚的映射，任何人来调试和维护这些代码时，都可以很容易地发现这些最容易发生逻辑错误的地方。

48. 为了使一个程序更强壮，每个组件都必须是很强壮的。在我们创建的类中运用 C++中提供的所有工具：隐藏实现、异常、常量更正、类型检查等等。用这些方法我们可以在构造系统时安全地转移到下一个抽象层次。

49. 建立常量更正。这允许编译器指出一些非常细微且难以发现的错误。这项工作需要经过一定的训练，而且必须在类中协调使用，但这是值得的。

50. 充分利用编译器的错误检查功能，用完全警告方式编译我们的全部代码，修改我们的代码，直到消除所有的警告为止。在我们的代码中宁可犯编译错误也不要犯运行错误（比如不要用变参数列表，这会使所有类型检查无效）。用assert()来调试，但要用异常来处理运行时错误。

51. 宁可犯编译错误也不要犯运行错误。处理错误的代码离出错点越近越好。尽量就地处理错误而不要抛出异常。用最近的异常处理器处理所有的异常，这里它有足够的信息处理它们。在当前层次上处理我们能解决的异常，如果解决不了，重新抛出这个异常。

52. 如果我们用异常说明，用set_unexpected() 函数安装我们自己的unexpected()函数。我们的unexpected()应该记录这个错误并重新抛出当前的异常。这样的话，如果一个已存在的函数被重复定义并且开始引起异常时，它不会不引起整个程序中止。

53. 建立一个用户定义的terminate()函数（指出一个程序员的错误）来记录引起异常的错误，然后释放系统资源，并退出程序。

54. 如果一个析构函数调用了任何函数，这些函数都可能抛出异常。一个析构函数不能抛出异常（这会导致terminate()调用，它指出一个程序设计错误）。所以任何调用了其他函数的析构函数都应该捕获和管理它自己的异常。

55. 不要自己创建私有数据成员名字“ 分解 ”，除非我们有了许多已有的全局值，否则 让类和命名空间来为我们做这些事。

56. 如果我们打算在for循环结束之后使用一个循环变量，要在 for控制表达式之前定义这个变量，这样，当 for控制表达式中定义的变量的生命期被限制在 for循环之内时，我们的程序依然正确。

57. 注意重载，一个函数不应该用某一参数的值来决定执行哪段代码，如果遇到这种情况，应该产生两个或多个重载函数来代替。

58. 把我们的指针隐藏在包容器类中。只有当我们要对它们执行一个立即可以完成的操作时才把它们带出来。指针已经成为出错的一大来源，当用 new运算符时，应试着把结果指针放到一个包容器中。让包容器拥有它的指针，这样它就会负责清除它们。如果我们必须有一个游离状态的指针，记住初始化它，最好是指向一个对象的地址，必要时让它等于 0。当我们删除它时把它置0，以防意外的多次删除。

59. 不要重载全局new和delete，我们可以在一个类跟随类的基础上去重载它们。重载全局new和delete会影响整个客户程序员的项目，有些事只能由项目的创建者来控制。当为类重载new和delete时，不要假定我们知道对象的大小，有些人可能是从我们的类中继承的。用提供的参数，如果我们做任何特殊的事，要考虑到它可能对继承者产生的影响。

60. 不要自我重复。如果一段代码在派生类的许多函数中重复出现，就把这段代码放在基类的一个单一的函数中然后在派生类中调用它。这样我们不仅节省了代码空间，也使将来的修改容易传播。这甚至适用于纯虚函数（见第 14章）。我们可以用内联函数来提高效率。有时这种相同代码的发现会为我们的接口添加有用的功能。

61. 防止对象切片。实际上用值向上映射到一个对象毫无意义。为了防止这一点，在我们的基类中放入一些纯虚函数。

62. 有时简单的集中会很管用。一个航空公司的“旅客舒适系统”由一系列相互无关的因素组成：座位、空调、电视等等，而我们需要在一架飞机上创建许多这样的东西。我们要创建私有成员并建立一个全部的接口吗？不，在这种情况下组件本身也是公开接口的一部分，所以我们应该创建公共成员对象。这些对象有它们自己的私有实现，所以也是很安全的。

附录C　模拟虚构造函数

在一个构造函数调用期间，虚机制并不工作（出现早期绑定），有时这很让人为难。

另外，我们可能想组织我们的代码，使得在创建一个对象时不用选择一个确切的构造函数类型，也就是，我们想说：“我不准确地知道我们是哪种类型的对象，但这与我们自己创建自己的信息有关。”这个附录介绍了两种“虚构造函数”方法。第一个是一种充分发展的技术，它在堆栈上和堆上都可以工作，但实现起来较为复杂。第二种则简单得多，但我们只能在堆上创建对象。

C.1　全功能的虚构造函数

让我们来看一个经常被引用的“shapes”例子。对于一个对象，我们可能想在构造函数内设置每件事，然后调用draw()来画出这个对象。draw()应该是一个虚函数，是一条它应该准确画它自身的消息，依赖于它是一个圆、矩形、线等。然而，这些在构造函数内并不能工作，原因就是第14章介绍的：当在构造函数中调用一个虚函数时，虚函数将分解为本地的函数体。

如果我们想在构造函数中调用一个虚函数并让它能正确工作，我们必须用一种模拟虚构造函数的技术。这是一个棘手的问题。记住虚函数是这样一种函数：我们送一条消息到一个对象，然后让这个对象算出该做些什么。但构造函数用来创建一个对象。所以一个虚构造函数好象在说：“我不确切地知道你是哪种类型的对象，那么你自己创建吧。”在一般的构造函数中，`编译器必须知道哪个VTABLE的地址绑到了这个VPTR上和它是否已存在，一个虚构造函数并不能做这些，因为它在编译时并不知道全部的类型信息。因此说一个构造函数不能是虚函数是有道理的，因为它是一种必须知道对象类型全部信息的函数。

仍然有一些时候，我们想要一些东西类似于虚构造函数的行为。

在shape这个例子中，在构造函数的参数表中传递一些特殊信息，并让构造函数在没有更多信息的情况下创建特殊的形状（圆、矩形或三角形）。通常，我们不得不显式地调用circle、square或triangle的构造函数。

Coplien [1] 称他对这个问题的解决方案为“信封和信纸”类。信封类是基类，一个包含指向基类对象的指针的外壳。“信封类”的构造函数决定（在运行时间，即当构造函数被调用时，而不是编译时间，即当类型检查已完成时）创建哪种特定类型，然后创建这种类型的对象（在堆上），并将这个对象赋给它的指针。所有的函数调用都由基类通过它的指针来完成。

下面是shape例子的一个简化版：

```
//: SSHAPE.CPP -- "Virtual constructors"
// Used in a simple "shape" framework
#include <iostream.h>
#include "..\14\tstash.h"

class shape {
```

[1] James O.Coplien，《高级C++编程风格与习语》，Addison_Wesley, 1992。

```
  shape* S;
  // Prevent copy-construction & operator=
  shape(shape&);
  shape operator=(shape&);
protected:
  shape() { S = 0; };
public:
  enum type { Circle, Square, Triangle };
  shape(type);  // "Virtual" constructor
  virtual void draw() { S->draw(); }
  virtual ~shape() {
    cout << "~shape\n";
    delete S;
  }
};
class circle : public shape {
  // Prevent copy-construction & operator=
  circle(circle&);
  circle operator=(circle&);
public:
  circle() {}
  void draw() { cout << "circle::draw\n"; }
  ~circle() { cout << "~circle\n"; }
};

class square : public shape {
  // Prevent copy-construction & operator=
  square(square&);
  square operator=(square&);
public:
  square() {}
  void draw() { cout << "square::draw\n"; }
  ~square() { cout << "~square\n"; }
};

class triangle : public shape {
  // Prevent copy-construction & operator=
  triangle(triangle&);
  triangle operator=(triangle&);
public:
  triangle() {}
  void draw() { cout << "triangle::draw\n"; }
  ~triangle() { cout << "~triangle\n"; }
};

shape::shape(type t) {
```

```
  switch(t) {
    case Circle: S = new circle; break;
    case Square: S = new square; break;
    case Triangle: S = new triangle; break;
  }
  draw();  // Virtual call in the constructor
}

main() {
  tstash<shape> shapes; // Default to ownership
  cout << "virtual constructor calls:" << endl;
  shapes.add(new shape(shape::Circle));
  shapes.add(new shape(shape::Square));
  shapes.add(new shape(shape::Triangle));
  cout << "virtual function calls:" << endl;
  for(int i = 0; i < shapes.count(); i++)
    shapes[i]->draw();
  shape c(shape::Circle); // Can create on stack
}
```

基类shape包含一个指向shape类型对象的指针，这也是它唯一的数据成员。当我们建立了一个“虚构造函数”配置时，我们必须确保这个指针已经被初始化为指向一个已存在的对象。

type枚举在类shape内部和shape的构造函数要在所有的派生类之后定义是这种方法的两个限制。每次我们从shape类中派生出新的子类型，我们必须回过头来把这个类型的名字加到 type枚举列表中。然后我们必须修改 shape的构造函数以处理新情况。其不利的一面是现在 shape类和所有 shape的派生类之间存在着依赖关系。有利的一面是这种一般情况下出现在程序体中（可能不止一处，这使得可维护性差）的关系被限制在一个类中。另外这还产生了一种像第 3章中介绍的“ cheshire cat ”的效果。在这种情况下，所有特定形状类的定义可以隐藏在实现文件中。这样，基类的接口是用户可以看到的唯一的东西。

在这个例子中，我们必须传递给构造函数的关于创建对象类型的信息是很清楚的，它是类型的一个枚举值。当然我们可以用其他信息——比方说，在语法分析程序中扫描器的输出可能被传递给“虚构造函数”，然后用这个文本串来决定准确的是要创建什么。

“虚构造函数”shape（type）只能在类中声明，它只有到这个基类的一切都声明之后才能定义。当然在class shape内可以定义缺省构造函数，但这应该声明为 protected，这样就不能产生临时shape对象。这种缺省构造函数只能被子派生类的构造函数调用。我们必须显式地产生缺省构造函数，因为只有在没有定义构造函数时编译器才能为我们自动产生一个缺省构造函数。因为我们定义了shape(type)，所以我们也必须定义shape（）。

在这个例子中，缺省构造函数至少有一个非常重要的工作——它必须将S指针的值置为0。这乍听起来很奇怪，但记住，缺省构造函数将作为创建实际对象——用Coplien的话来说是“信纸”，不是“信封”——的一部分来调用。因为“信纸”是从信封继承来的，所以它也继承了数据成员S。在“信封”中，S是很重要的，因为它指向一个实际的对象，但在“信纸”中，S只是一个多余的口袋。然而即使是多余的口袋，也应该初始化，如果“信纸”在调用缺省构造函数时没有把S置0，事情将变得很糟糕（后面我们将看到）。

“虚构造函数”完全由它的参数来决定对象的类型。虽然这种类型信息直到运行时才会被读取来起作用，而一般情况下，编译器在编译时就必须知道确切的类型（这是这个系统有效地模拟虚构造函数的另一个原因）。

在虚构造函数里有一个switch语句，这里用参数来构造实际的对象，然后将它指定给“信封”里的指针。在这之后，“信纸”创建就完成了，所以任何虚调用都会被正常地引导。

作为一个例子，我们来看在虚构造函数中调用 draw()，如果跟踪这个调用（手工或用一个调试器），我们可以看到它是从基类 shape中的draw()函数开始的，这个函数用“信封”中S指针指向的“信纸”的 draw()。所有从shape中派生的类都共享同一接口，所以这个虚调用被正确地执行，即使它似乎是在一个构造函数内部（实际上这个“信纸”的构造函数已经完成）。只要基类中所有的虚函数的调用都通过指向“信纸”的指针来完成的话，系统就会正确运行。

为了理解它是如何工作的，现在来看 main()中的代码。为了产生数组 s[]，“虚构造函数”shape被调用。一般在这种情况下，我们就要调用实际类型的构造函数，而这个类型的 VPTR应当被安装在这个对象中。然而现在，每种情况下使用的 VPTR都是shape类的，而不是特定的circle、square 或triangle类的。

在for循环中，每个shape调用draw()函数，这个虚函数通过VPTR来决定相应的类型。当然这里每个情况下都是 shape。实际上，我们可能奇怪为什么 draw被做成完全虚的，下一步显示了其理由：基类的draw()函数通过“信纸”指针S来调用了“信纸”的draw()函数。这时函数调用决定针对实际的对象类型，而不仅是基类 shape。因此，使用“虚构造函数”的运行时间支出比我们每次调用虚函数时要多。

剩下的问题

这里所讲述的“虚构造函数”中，析构函数内部的虚函数调用是在编译时用一般的方法确定的。我们可能认为，通过在析构函数中巧妙地调用基类的虚函数，我们可以回避这种限制。不幸的是，这会导致灾难性的结果。基类中的函数确实要调用。然而它的 this指针是指向“信纸”而不是“信封”的。所以当基类的函数被调用时——记住，基类中的虚函数总是以为它正在处理“信封”——它将用S指针来完成调用。对于“信纸”，S是0，由protected的缺省构造函数设置。当然我们可以通过在基类中的每个虚函数中加入一段代码来检查 S是不是为0，从而防止上述0指针调用，或者我们在使用“虚构造函数”时遵守下面两条原则：

1) 在派生类函数中不要显式地调用根类的虚函数。

2) 总是重定义根类中所有的虚函数。

这两条规则起因于同样的问题，如果我们在一个“信纸”内调用一个虚函数，这个函数获得“信纸”的this指针。如果虚函数没有重载，这个调用将调用基类中的虚函数，然后调用将通过“信纸”的this指针指向“信纸”的S，而这又是0。

我们可以看出，用这种方法费时、受限而且危险，这就是为什么不想让它在所有时候都作为编程环境的一部分的原因。它是属于那种解决某些问题时非常有用，但我们不想在所有时候都运用的特征。幸运的是，C++允许我们在需要它时，再把它放入，而标准的方法是最安全的，最终也是最容易的。

• 析构操作

析构操作也很棘手。为了便于理解，让我逐句跟踪当我们对于指向创建在堆上的 shape对象的指针调用delete时——特别是对于一个square时——会发生什么。（这比在堆栈中创建的对

象更复杂)。这个delete将穿越多态接口，就像在main()中的delete s [i]语句一样。

指针S[i]的类型是基类shape，所以编译器让所有的调用都通过shape类。一般情况下，我们可能说这是虚调用，所以square的析构函数将被调用。但用这种“虚构造函数”方案时，编译器正在产生一个实际的shape对象，即使构造函数初始化“信纸”的指针指向一特定类型。虚机制被使用，但shape对象中的VPTR是shape的VPTR，而不是square的。这样就决定了对应于shape的析构函数，它对于“信纸”的指针S 调用delete，而S实际指向一个square对象。这又是一个虚调用，但这次是调用square的析构函数。

当然在继承层次中的所有析构函数都会被调用， square的析构函数首先被调用，紧接着是其他中间的析构函数，按次序，一直到基类的析构函数被调用。在基类析构函数中有 delete S语句。当这个函数最初被调用时，它是为“信封” S的，但现在它是为“信纸” S，这是因为“信纸”是从“信封”中继承过来的，而不是因为它包含了什么。因此这个 delete调用不做任何事。

为了解决这个问题，可以让“信纸”的 S指针等于零。然后当“信纸”基类析构函数被调用时，得到delete 0，它被定义为不做任何事。因为缺省构造函数是protected，它只能在“信纸”的构造期间被调用，所以这是将S置零的唯一情况。

C.2 更简单的选择

“虚构造函数”对于大多数应用来说并不一定很复杂。如果我们能限制我们自己只在堆上产生对象，事情可以变得简单得多。我们所需要的是某个函数（我把它称为对象制造函数），我们可以向它发送一串信息，它将会产生正确的对象。如果它可以产生一个基类指针而不是对象本身，那么这个函数不一定是构造函数。下面是用对象制造函数重新设计的“信封——信纸”例子：

```
//: SSHAPE2.CPP -- Alternative to SSHAPE.CPP
#include <iostream.h>
#include "..\14\tstash.h"

class shape {
  shape(shape&); // Prevent copy-construction
protected:
  shape() {} // Prevent stack objects
  // But allow access to derived constructors
public:
  enum type { Circle, Square, Triangle };
  virtual void draw() = 0;
  virtual ~shape() { cout << "~shape\n"; }
  static shape* make(type);
};

class circle : public shape {
  circle(circle&); // No copy-construction
  circle operator=(circle&); // No operator=
protected:
  circle() {};
```

```
public:
  void draw() { cout << "circle::draw\n"; }
  ~circle() { cout << "~circle\n"; }
  friend shape* shape::make(type t);
};

class square : public shape {
  square(square&); // No copy-construction
  square operator=(square&); // No operator=
protected:
  square() {};
public:
  void draw() { cout << "square::draw\n"; }
  ~square() { cout << "~square\n"; }
  friend shape* shape::make(type t);
};

class triangle : public shape {
  triangle(triangle&); // No copy-construction
  triangle operator=(triangle&); // Prevent
protected:
  triangle() {};
public:
  void draw() { cout << "triangle::draw\n"; }
  ~triangle() { cout << "~triangle\n"; }
  friend shape* shape::make(type t);
};

shape* shape::make(type t) {
  shape* S;
  switch(t) {
    case Circle: S = new circle; break;
    case Square: S = new square; break;
    case Triangle: S = new triangle; break;
  }
  S->draw(); // Virtual function call
  return S;
}

main() {
  tstash<shape> shapes; // Default to ownership
  shapes.add(shape::make(shape::Circle));
  shapes.add(shape::make(shape::Square));
  shapes.add(shape::make(shape::Triangle));
  cout << "virtual function calls:\n";
  for(int i = 0; i < shapes.count(); i++)
```

```
    shapes[i]->draw();
//Circle c; // error: can't create on stack
}
```

可以看到，所有的事情都变得简单了，而且不用担心内部指针引起的奇怪情况。唯一的限制是基类中的enum在我们每次派生一个新类时都必须更改（用信封——信纸的方法也是如此），而且对象必须在堆中创建。这后一条是通过使所有构造函数为 protected来强制完成的，所以用户不能创建这个类的对象，但派生类的对象在对象创建期间可以访问基类的构造函数。这是protected关键字不可缺少的一个很好的例证。